ADVANCED DISTRIBUTED SIMULATION TECHNOLOGY

ModSAF

SOFTWARE ARCHITECTURE DESIGN

AND OVERVIEW DOCUMENT

Ver 1.0 - 20 December 1993

CONTRACT NO. N61339-91-D-0001

D.O.: 0021

CDRL SEQUENCE NO. A004

Prepared for:

U.S. Army Simulation, Training, and Instrumentation Command (STRICOM)
12350 Research Parkway
Orlando, FL 32826-3276

Prepared by:

LORAL
Systems Company

ADST Program Office
12151-A Research Parkway
Orlando, FL 32826

REPORT DOCUMENTATION PAGE

Form approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	12/20/93	

4. TITLE AND SUBTITLE
ModSAF SOFTWARE ARCHITECTURE DESIGN AND OVERVIEW DOCUMENT

5. FUNDING NUMBERS
C N61339-91-D-0001, Delivery Order (0021), ModSAF (CDRL A004)

6. AUTHOR(S)
Dr. Andy Ceranowicz, Carol Ladd, Joshua Smith, Robert Vrablik,

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Loral Systems Company
ADST Program Office
12151-A Research Parkway
Orlando, FL 32826

8. PERFORMING ORGANIZATION REPORT NUMBER
ADST-TR-W003267

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Simulation Training and Instrumentation Command (STRICOM)
12350 Research Parkway
Orlando, FL 32826-3275

10. SPONSORING ORGANIZATION REPORT
ADST-TR-W003267

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE
A

13. ABSTRACT (Maximum 200 words)
This document provides information regarding architectural features of the ModSAF application. Various sections discuss the basic concepts of ModSAF, the physical and behavioral models, architectural support, functional components, design methodology, and the procedures for extending ModSAF with new objects or behaviors. The last chapter steps the reader through the process of defining, building, and executing new capabilities in ModSAF.

14. SUBJECT TERMS
Modular Semi-Automated Forces, DIS, ADST, BDS-D

15. NUMBER OF PAGES
Approx 100

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	17. SECURITY CLASSIFICATION OF THIS PAGE	17. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

1 Overview

Modular Semi-Automated Forces (ModSAF) is a Computer Generated Forces (CGF) system for creating and controlling entities on a simulated battlefield. ModSAF simulated entities can behave autonomously; they can move, shoot, communicate, and react without operator intervention. These entities can interact with each other and with manned simulators on the network. Since ModSAF provides a realistic representation of forces without requiring a large number of personnel, it can supplement a simulation exercise's capability to conduct force-on-force engagements and can permit large scale, combined arms team training within a distributed interactive simulation (DIS) environment.

The goal of ModSAF is to replicate the outward behavior of simulated units and their component vehicle and weapon systems to a level of realism sufficient for training and combat development.A ModSAF entity is given extensive capabilities; it can drive over the terrain avoiding obstacles, shoot at enemy objects, and be tasked to execute a mission.

ModSAF simulates an extensive list of entities. For fixed wing aircraft, it simulates the F-14D, MIG-29, A-10 and SU-25. For rotary wing aircraft, it simulates the AH-64, OH-58D, Mi-24, and Mi-28. For its ground forces, ModSAF can simulate tanks (M-1 and T-72), infantry fighting vehicles (M-2 and BMP), ADA (ZSU-23/4), and dismounted infantry. Enhancements could result in the support of additional physical models such as Cavalry, Howitzers, Mortars, Minefields, CSS, Scud, and Patriot.

At the vehicle/weapons system level, ModSAF simulates entities by giving them the capability to execute a realistic range of basic actions inherent to the entity type. For example, a simulated tank can drive along a road, slew its turret, and turn in place. A simulated airplane can take off, orbit, and land. All weapons systems exhibit realistic rates of fire and realistic trajectories.

ModSAF simulated entities can exhibit mobility, firepower, and catastrophic combat damage when hit by enemy fire. Their resources (both fuel and ammunition) are accurately depleted as they move and shoot. Other simulated capabilities include intervisibility, target detection, target identification, target selection, fire planning, and collision detection. These capabilities are based on, but are not limited to, such appropriate factors as range, motion, activity, visibility, direction, orders, and evaluation of threat.

When a unit is simulated, ModSAF not only creates the ModSAF entities in a unit (such as a plane or tank), but also builds a structure corresponding to the unit hierarchy. Commands can then be issued to the top-level units or to subordinate units or vehicles. ModSAF interprets orders and generates the appropriate unit and vehicle behavior and tactics.

The automated behavior performed by a ModSAF entity or unit is governed by *tasks*. Examples of tasks include behavior such as move, orbit, avoid collisions, and search for enemy vehicles. These tasks are typically implemented as augmented finite state machines. ModSAF uses a set of representative tasks for both individual (vehicle) or collective (unit) tasks. These tasks are defined in terms of their characteristic parameters. Although ModSAF relies on standard military doctrine to supply default values for task parameters, the ModSAF user is allowed to modify the default values.

ModSAF groups a set of related tasks that run at the same time into a *task frame*. A task frame is typically composed of moving, targeting, and reacting tasks that all work together to accomplish the frame's goal. Some examples of task frames are Move, Halt, Assault, and Return to Base.

A ModSAF vehicle or unit can be commanded to execute a *mission* which is a collection of one or more task frames with each task frame representing one mission component or phase. The user is allowed to specify the condition(s) that must be satisfied to permit transitioning between mission phases.

ModSAF generates multiple entities that can be controlled by a single operator. The number of entities is maximized by simulating only those features that are externally observable or significant to other simulation exercise participants, and by automating the low-level decision making of the entities. However, ModSAF lets the operator make the critical tactical decisions for the forces they control by allowing them to override or interrupt any automated behavior.

The ModSAF architecture is both flexible and hierarchical. It allows a researcher to embed other behavioral representations within the architecture, and it provides support for explanation, inspection, and modification of behavior.

2 Software Architecture Introduction

This section introduces the Software Architecture of ModSAF. It is divided into two sections; the first section introduces the basic concepts of ModSAF, the next section describes the ModSAF main loop.

2.1 Basic Concepts

2.1.1 Semi-Automated Forces

Semi-Automated Forces (SAF) are computer systems for simulating and controlling entities, such as vehicles, Dismounted Infantry (DI), missiles and dynamic structures. These entities are used to populate the virtual battlespace. SAF entities supplement the manned simulators by creating realistic and robust scenarios for soldiers at a reasonable cost. These entities can perform opposing, flanking, subordinate, and supporting force roles. The number of entities simulated is greatly increased by carefully simulating only those features that are externally visible and/or significant to the other simulation participants. Control of a large number of entities is achieved by simulating the low level decision logic of the entities and placing the operator in supervisory control of them. The operator controls his forces by issuing operations orders and radio fragos that augment the built in automated reactions of the SAF forces. This man-in-the-loop approach provides interesting and adaptive opponents without difficulty and computational expense of full automation. The SAF operator intercedes in those situations where the automated logic lacks an adequate response.

2.1.2 The SAFSim SAFstation and SAFlogger

The ModSAF architecture divides its functions into three components: the ModSAF Command Station or SAFstation, the ModSAF Simulator or SAFsim, and the ModSAF Exercise Logger or SAF-logger. These components are typically run on separate computers distributed over a network (although the SAFsim and SAFstation can run on the same computer). The components communicate physical battlefield state and events between themselves via the DIS protocol and command, control, and system information via the Persistent Object Protocol. Database abstractions are used to simplify the interfaces of and make uniform for both distributed and local components. The database abstractions for the DIS database are objects like entity state, impact and collision. The database abstractions for the PO protocol are objects such as control measures, units and

tasks. The recommended hardware for ModSAF is configured to be able to run any ModSAF component. An exercise using ModSAF can be configured in a variety of ways making optimal use of the hardware for that application. The SAFsims work together to provide a simulation server that responds to commands from users at SAFstations. For exercises requiring large numbers of entities you would allocate more computers to run as SAFsims increasing your simulation resources. One user at one SAFstation can control a hierarchy of SAFsims to simulate large units with many entities. For exercises that require low level human control of SAF entities or human operator interaction with manned simulators via radio, you can allocate more computers to be SAFstations. Multiple users at multiple SAFstations can control entities simulated on one or many SAFsims.

2.1.3 The DIS Protocol

The Distributed Interactive Simulation (DIS) protocol is used to create a distributed simulation environment where battlefield entities simulated on different processors can interact. DIS can be thought of as a set of protocols for linking simulators together, as the software for implementing the protocols on various computers, and as a common, consistent, shared, simulated world where differents types of simulators can interact.

The basic concepts of Distributed Interactive Simulation (DIS) are an extension of the Simulation Networking (SIMNET) program developed for the Defense Advanced Research Projects Agency (DARPA) by BBN. The purpose of DIS is to allow dissimilar simulators, distributed over a large geographical area, to interact in a team environment. Communications are over local and wide area networks.

Simulations currently used within DIS include manned vehicle simulators, dismounted infantry simulators, semi-automated forces (SAF) simulators, and command post simulators. Also used within DIS are tools for data collection and analysis, including a "stealth vehicle".

Semi-automated forces (SAF) simulators behave in the same manner as vehicle simulators in that they broadcast their states on the network. The types of information that DIS broadcasts for SAF vehicles includes the entity state, impact, collision, fire, initialization, radar and weather.

2.1.4 The PO Protocol

The Persistent Object (PO) protocol is a general mechanism for supporting a dynamically updated, robust, shared database among a group of simulators. The development of the protocol

was motivated by a desire to provide a more flexible and scalable interface between the components of Semi-Automated Force (SAF) systems.

In the ModSAF system, the Persistent Object Protocol enables all command and control information to be shared by and modified from any SAFstation. It also allows commands to be distributed from any SAFstation to any SAFsim for execution. This enables load leveling and flexibility in system configuration. The same architecture can support systems with relatively large numbers of SAFstations (e.g. for close supervisory control) or with relatively large numbers of SAFsims (e.g. in order to generate large numbers of DIS entities with a small human staff). Finally, it allows objects from failed SAFsim and SAFstation to be automatically taken over by a surviving system.

The basic properties of the PO Database are:

- Every Simulator has access to a local copy of the entire DB
- Every Simulator can create, change or delete objects, and can query and search the DB
- Every Simulator is notified when objects are created, changed or deleted
- DB is self-correcting in the face of packet loss on the network.

2.1.5 Modular Programming

ModSAF has a modular software architecture that encourages users to extend and modify the system to support their applications. Over 99% of the ModSAF software is implemented as library modules with strictly defined and documented public interfaces. Layering and callback techniques are user to minimize module interdependence. Small main programs link together these libraries to form ModSAF applications. The ModSAF architecture is object-based, dividing the world into distinct objects whose activities are simulated individually. All simulated entities are data driven so that the behavioral and physical models used to simulate them can be modified during scenario generation as well as at runtime. The representation of physical models is separated from that of behavioral models. Behavioral models are represented by task libraries and are executed by a task manager. Physical models are represented by sets of libraries that are interfaced to the simulation via generic interfaces. These generic interfaces are defined to allow models in the same family to be interchanged.

2.1.6 The Physical and Behavioral Models

A vehicle in ModSAF is described by a series of vehicle subclasses. Each of these subclasses is either a physical or behavioral model and is represented by a separate ModSAF library. For example, libcollision provides a 3D physical model of collision detection. libvspotter implements a behavioral model (or task) which accumulates detected vehicles from a vehicles sensor. One entity in ModSAF will be composed of many physical and behavioral models or vehicle subclasses.

All the simulation models which make up a particular vehicle are listed in a data parameter file. Each vehicle subclass in the data file lists the parameters for that vehicle subclass. Generic versions are generally written (such as tracked-hull), and they are customized for a particular vehicle (M1, M2, T72, T80, etc.) via their parameters. This is true not only of physical components, but also of behavioral descriptions and architectural support modules.

2.1.6.1 Physical Models

The physical models describe the physical characteristics of an entity. Physical models are represented by sets of libraries that are interfaced to the simulation via generic interfaces. These generic interfaces are defined to allow models in the same family to be interchanged.

Each physical model of an entity is invoked in ModSAF periodically to update the internal state of the model. The physical model library contains a tick routine that defines the periodic routine to call. This routine provides the vehicle the chance to update the internal state of this model.

2.1.6.2 Behavioral Models

In the ModSAF architecture, the behavioral aspects of a platform are encoded using *tasks*. Each task is a vehicle subclass encoded as a finite state machine. A special subclass called the task manager (libTaskMgr) is responsible for triggering the execution of these tasks based upon the list of behaviors represented in the C2 (PO) database (grouped into a *task frame*). A sketch of the current state of each executing task is maintained on the network, so that the platform executing the task may be seamlessly transferred from one ModSAF simulator to another.

The foundation of the ModSAF command and control framework is the idea of a *task*. Simply put, a task is a behavior performed by an individual on the battlefield. Tasks may be done on behalf of a unit (company road march) or they may directly control a physical system (drive toward a waypoint). Tasks may be done to achieve a mission objective (attack an objective), or they may be

done continuously, independent of the mission (scan for enemy). Tasks may be representations of actual battlefield behavior (run for cover), or they may be implementation details of the simulation (arbitrate between several possible alternative actions).

Exactly which portions of the simulation are represented with tasks is a matter of implementation. Whereas the initial ModSAF implementation represents almost all non-physical systems with tasks, other implementations could use tasks only to describe the mission, and use other representations for internal behaviors.

2.1.7 The User Interface

The graphical user interface(GUI) for ModSAF provides the user with a view of the simulated world. It also allows the user to change parts of the simulated world.

The architectural framework provides the top-level layout of the user interface. The user interface consists of a terrain map, menubars, buttons which place the user in various modes, a radio message log, and an editor area at the bottom of the screen which provide the user with a mechanism for modifying the simulated world.

2.1.8 Communications

The purpose of Distributed Interactive Simulation is to allow dissimilar simulators, distributed over a large geographical area, to interact in a team environment. Communications are over local and wide area networks. The job of ModSAF is to simulate SAF entities. These are broadcast across the network allowing them to work with other simulators on the same network.

In ModSAF, there are two types of protocols being used for simulation, the DIS protocol and the PO protocol. The DIS database details entity state, impact, collision, fire, initialization, radar and weather. The PO Database details control measures, units, tasks, task frames, missions and model parameters.

Both of these protocols are used to allow ModSAF to communicate across the network with other simulators. A ModSAF system sends out information on the entities it is simulating, and also listens for other simulations on the network. This together allows ModSAF to work with the complete DIS battlefield.

2.1.9 Architectural Support

The ModSAF software architecture allows multiple researchers and developers to combine separately designed and built modules into the same experiment.

The ModSAF software architecture is an extensible set of software modules which allows rapid development and testing of new agents in the DIS simulated environment. Many ideas for the command and control of automated DIS agents can be implemented and tested without extensive redevelopment of already available SAFOR supporting code.

To be successful, the ModSAF architecture must be able to evolve as the knowledge about building SAFOR systems evolves and as new applications arise. Because there is still much to learn, the architecture is flexible and expandable while retaining backward compatibility.

ModSAF has the following architectural aspects:

- Allows replacement of individual subsystems without modification of the surrounding software;
- Defines programming practices which will ensure interoperability between independently developed subsystems;
- Allows the use of diverse hardware, which will minimize the buy-in cost to researchers by allowing them to use available hardware;
- Allows subsystems to be written in almost any computer language;
- Allows arbitrary distribution of subsystems across different hardware platforms at run time; and,
- Supports real time, faster than real time, and slower than real time simulation.

2.2 The ModSAF Main Loop

When ModSAF is initialized, it declares functions that need to be called periodically and what that period is. All these functions are time-sliced using a scheduler. Some functions need to be called more often that others. So for instance, the scheduler may process some user interface requests, then simulate a vehicle, then update the map, then check for input from the command line interface, then check for some more user interface requests, etc.

The scheduler is the main loop executed in ModSAF. It schedules all the ModSAF functions that are called. The scheduler calls functions either periodically, or once after a specified delay. ModSAF defines a number of rings in the scheduler, each ring is defined by how often the functions

in the ring should be called and what those functions are. The scheduled functions are invoked at a rate that is no faster than that specified for its ring. Each function that is called periodically is referred to as a tick routine.

The scheduler uses the realtime and simulation clocks to schedule function calls. The realtime clock advances with the system clock. The simulation clock uses the realtime clock to advance. The difference with the simulation clock is that time is frozen within the context of one tick for a vehicle. In other words, the tick routines that are called for a specific vehicle all assume the same time even though they are called at different realtimes. This is to ensure that all the routines that tick for a vehicle are happening at the same time. The simulation clock can also be configured to proceed faster or slower than the realtime clock.

Currently in ModSAF, the following tick routines exist for the following periods:

```
period  tick routine
10      tick_X
                processes all events in the X queue
67      pv_read_loop
                reads packets from the network
        safparse_tick
                reads from the command line interface
        prev_tick
                ticks a previewer if any were started
        local_tick
                ticks all the vehicle subclasses
        remote_tick
                calls:
                ent_tick
                        for remotes, manages timeouts and RVA,
                        for locals, sends appearance if warranted
                        by RVA thresholding
                dsg_tick
                        ticks the designator
                stealth_tick
                        ticks the stealth
                pvd_tick
                        ticks the Plan View Display
250     preview_tick
                        ticks the previewer
1000    tick_assoc
                        ticks the association layer
5000    vw_update_viewable
                        ticks the list of viewable
```

When ModSAF is running it sits in a constant scheduler loop which just figures out which

routines to run. If a routine at a certain period is not ready to be run, then the scheduler will call a routine in a lower period ring, or if nothing is ready in that period ring a lower period ring, etc, until the routine in the original period ring is ready to run.

3 Functional Components of ModSAF

The five major components of ModSAF are the architectural support, the physical models, the behavioral models, the user interfaces, and the data logger. The following sections provide a more detailed description of each of these components.

3.1 Architectural Support

3.1.1 Modular Library Structure

A class is defined as a group, set, or kind that shares common attributes. The ModSAF architecture defines two classes: c2obj classes and safobj classes. c2obj (Command and Control Objects) classes are tasks, tasks frames, etc. These objects define command and control in the SAF battlefield. safobj (SAF Object) classes consist of vehicle models, and are commonly referred to as vehicle subclasses. `libclass` is the library that manages these ModSAF classes.

3.1.1.1 LibClass Overview

An *object* is defined as "an area in computer memory that serves as a basic structural unit of analysis" (Baron). A *class* defines the organization of data in that memory, and the functions which operate on a group of objects which use the same organization. Typically, a library will define a single class and will provide functions to create, destroy, or operate on objects in that class.

Often the same object is represented in two ways, once at a low software layer, and again in a higher layer. For example:

Object Type	Low-Layer Representation	High-Layer Representation
Simulated Entity	LibVTab vehicle	LibSAFObj object
Route	LibPO persistent object	LibC2Obj object

Various libraries define classes of objects which are instantiated. Most notably:

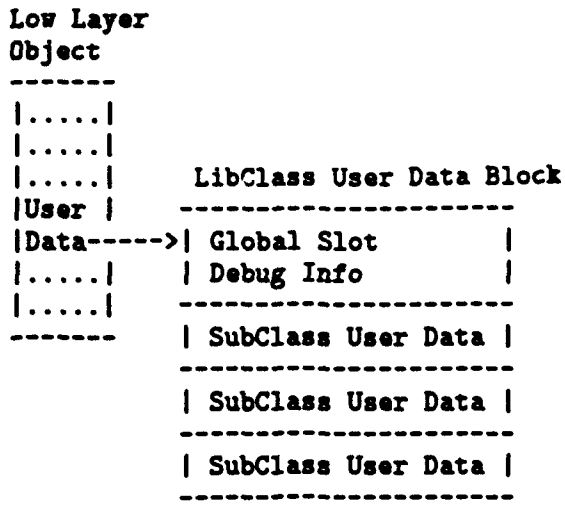
- `libpo` creates graphics, units, task frames, etc.
- `libvtab` creates vehicles.

- Xt creates widgets.

Different high-layer classes need to attach different user data to these objects. For example, when a route is created in libpo, the user interface software makes a bunch of widgets and stuff which it wants to attach as user data to the object. Simultaneously, the simulation software wants to compile the route into its internal format, and attach that as user data. This leads to an incompatibility which will prevent linking the workstation and simulation software together. What is needed is a way to declare at run time the number of pieces of user data that will be attached to each class of object.

Low layer classes generally allow the attachment of one piece of user data to each object (this is true of `libpo`, `libvtab`, and Xt). `libclass` provides a sort of user data multiplexer service to allow each class (each library) within an application to attach its own kind of user data to each object.

As shown in the figure below, a single slot is also provided for a 'global' piece of user data to be attached to each block. This slot is accessed through functions rather than the subclass slots.



libclass also provides debugging functions. It manages run time modifiable flags which enable or disable debugging for each class, on both a global and a per-instance basis, and it manages 'show' routines for each class.

3.1.2 Command and Control Classes

Several classes of objects are defined specifically for command and control. The superstructure that defines these subclasses is located in **libc2obj**. The various subclasses are defined in individual libraries, which are documented below.

3.1.2.1 Tasks

A Task object represents an individual behavior of SAF vehicles or units, such as avoid collisions, go to point, or follow road. Task behaviors are typically implemented as finite state machines using the **AAFSM** code generator. A Task State object is used to maintain any public context for the task.

libtask manages the association between Task class objects in the PO database and the vehicle subclasses which execute those tasks.

3.1.2.2 Task Frames

A Task Frame object groups a collection of zero or more tasks which execute in parallel. Task frames are pushed and popped off a stack as behaviors are added and resumed in the course of execution. At the bottom of the stack is the background task frame, which contains the tasks that dictate the default behavior of the unit. `libtaskmgr` is a C2 subclass that runs all the tasks in the current frames for a SAF vehicle.

Special tasks called Enabling Tasks are also located in task frames. Enabling tasks are just predicate functions that monitor for conditions that will trigger them to start the frame in which they live. They are only run when their frame is not active.

3.1.2.3 Graphics

There are several command and control (C2) subclasses that represent graphical objects. They include PointClass, LineClass, SectorClass, and TextClass. `libgraphics` implements the display and editing of persistent objects. It handles the display of points, lines, text, and task frames, and the editing of points, lines, and text. Each class of object has a corresponding sensitive class that is used when the graphic is displayed. `libgraphics` also allows other libraries to register their own sensitive classes, which are used when the displaying text associated with objects of other classes (such as Units).

3.1.2.4 Units

UnitClass objects represent simulated entities. They store information pertaining to the unit capabilities, organization, markings, and various bookkeeping data. `libunits` manages the editing of unit class persistent objects. The display of these objects performed via `libbgrdb` icons.

3.1.2.5 Unit Organization

`libunitorg` manages unit organization information within the SAF simulation. It tracks both the task organized and functionally organized superior and subordinate relationships of units, in order to provide this information without the need for frequent persistent object database queries. `libunitorg` also tracks changes to units that are made by outside sources (such as the GUI), and updates the simulation accordingly.

3.1.2.6 Overlays

An overlay is an individually colored layer containing graphics. Overlays have name and color attributes, and are used to group other objects together for display purposes. **liboverlay** (and **liboeditor**) manage the creation and editing of overlays. By enabling and disabling individual overlays, only those graphics of current interest need be displayed.

3.1.3 Vehicle Tables

The need to build lists of vehicles is common in ModSAF software. A global list is needed at the top level to keep track of all the entities in the simulated environment. In addition, particular functions, such as the detection model or targeting, often need to keep lists of subsets of entities (such as a list of detected vehicles, or a target list). The global list of vehicles is created and maintained by **libvtab**, while the position-based list is managed by **libpbtab**. One reason for the position-based table is to efficiently determine which vehicles were damaged by a bomb that exploded at a particular location on the terrain. The following sections describe each type of vehicle table.

3.1.3.1 VTab

libvtab assigns an integer identifier to vehicles in the simulated environment. This id can be used to easily reference each vehicle. **libvtab** also provides a class (data structures and functions) which can be used to create lists of vehicles using a variety of data structures.

libvtab uses a sequential map to implement its master vehicle table. Each vehicle is given a number (derived from its network VehicleID) which corresponds to its location in this map. Thereafter, the vehicle is referred to by its number, and a simple array reference will find the particular information. This integer reference method provides an easy way to get to vehicles in a symbolic debugger, and to refer to particular vehicles by number through the parser ("vehicle 1021 show driver").

Examples of the information that the UserData "user data" field can hold include anything the library needs to store on a per-vehicle basis: the time a vehicle was first sighted, the range to that vehicle, an index into a local array assigned to that vehicle, etc. In ModSAF, the UserData holds a pointer to a block of class data. This block is implemented as an array of pointers to the vehicle's subclass data which is typically held in a library's XX_VARS structure (such as COLL_VARS for libcollision).

The vehicle table sequential map is huge to accommodate future needs. It has slots for up to 65,535 total vehicles (id 0 is invalid). Given this large map size, it is not practical to use sequential maps for other vehicle lists. Instead, libvtab allows other classes (libraries) to create lists of vehicles using a variety of efficient data structures:

- Doubly Linked Lists
- Binary Trees
- Ordered Arrays

Refer to (see section "libvtab" in *LibVTab Programmer's Manual*). for more information.

3.1.3.2 PBTab

By using position-based vehicle tables, the computation required of each vehicle does not increase as the number of vehicles increases. The total amount of computation is not in proportion to the square of the number of vehicles. Vehicles are kept sorted by location, rather than by ID number. A query can return all vehicles in a given area.

`libpbtab` provides the position based vehicle table functionality using a set of data structures suited to present SAF needs. The table structure used is a sparsely filled quadtree. Refer to (see section "libpbtab" in *LibPBTab Programmer's Manual*). for more information.

3.1.4 Parameter Files

Parameter files provide the ModSAF application with various pieces of information such as vehicle characteristics, behavioral model parameters, and information concerning the setups of various editors (fields, default values, etc.). Parameter files are read in by the ModSAF application at run time so they can be modified without having to recompile ModSAF. This makes changing parameters easier and less time consuming for the user. Parameter files are characterized by their `.rdr` extension. The following sections describe parameter files in more detail.

3.1.4.1 Overview

There are three major groups in which most parameter files may be categorized: vehicle parameter files, editor specifications, and databases.

3.1.4.2 Vehicles

The vehicle parameter files are located in src/ModSAF/entities. They contain specifications for all simulated entities (tanks, wheeled vehicles, dismounted infantry, fixed-wing aircraft, rotary-wing aircraft, missiles, etc.).

Here is an example of a vehicle parameter file for a USSR Sagger missile:

```
USSR_Sagger_MODEL_PARAMETERS {

(SM_PBTab)
```

```
(SM_Entity (length_threshold 10.0)
           (width_threshold 10.0)
           (height_threshold 10.0)
           (rotation_threshold 3.0)
           (turret_threshold 3.0)
           (gun_threshold 3.0)
           (vehicle_class vehicleClassSimple)
           (guises munition_USSR_Sagger munition_US_TOW)
           (send_dis_deactivate false))

(SM_Collision (check buildings platforms ground trees)
              (announce buildings)
              (duration 0)
              (feature_mass 10000.0)
              (fidelity high))

(SM_Components (hull SM_MissileHull SAFCapabilityMobility))

(SM_MissileHull (sensor_name gunner-sight)
                (sensor_on_board false)
                (parent_sensor_name "")
                (pursuit_mode lead_pursuit)
                (simulation munition_USSR_Sagger)
                (range 3000.0)
                (launch_speed 5.0)
                (acceleration 250.0)
                (safe_time 0.25)
                (loal_time 0.1)
                (max_burn_time 20.0)    ; back calc'd fm speed/range.
                (burn_max_turn 5.0)
                (coast_max_turn 5.0)
                (directionality 12.566370614359)
                ;; 150 m/s
                (max_speed (0.0         0.49)
                           (20000.0     0.49))
                )
}
```

This file specifies various characteristics which describe how the entity behaves. The data for all entities is read by the ModSAF application at run time, and the libraries that need this information have access to it. The first element in each of the lists (i.e. SM_Entity, SM_Collision, etc.) generally indicate which library reads the information in that list.

There are many vehicle parameter files, and there are some behaviors and characteristics which do not vary over a group of entities. For example, the visual characteristics (which describe things like visibility range, haze factors, and tree opacity) do not vary from vehicle to vehicle. Thus,

instead of specifying the same data in many parameter files, such data is specified once in one parameter file, and is then accessed from each vehicle parameter file. This saves time and space.

There are currently three levels of specificity in the vehicle parameter files. They are, from the general to the specific:

- standard_params.rdr
- macros.rdr
- The individual vehicle .rdr files.

When adding a parameter, you must determine in which of the above three files it belongs. In general, this set of rules may be followed:

- If the parameter has the same value(s) for all vehicles of a category (the categories being ground vehicles, fixed wing aircraft, rotary wing aircraft, and missiles), then it belongs in **standard_params.rdr**.
- If the parameter has the same value(s) for many vehicles, but not all vehicles of a category, then make a macro out of it, put the macro in **macros.rdr**, and refer to the macro in the appropriate vehicle parameter files. Refer to (see section "libreader" in *LibReader Programmer's Manual*). for more information about macros.
- If the parameter is different for most vehicles, then it is best to specify it in each vehicle parameter file.

When looking for a parameter, start at the individual vehicle parameter file. If it's not there, look in **macros.rdr**. If it's not there, look in **standard_params.rdr**.

3.1.4.3 Editor Specifications

Some parameter files are editor definitions, which define the pieces of data that the user may specify from various editors in the GUI. Editor parameter files specify the editor title, names of editor fields, which data type these fields are (i.e., string, integer, floating point values), whether they're optional or required, and initial values. Here is an example of an editor parameter file. It describes the **utraveling** editor.

```
;;
;; $RCSfile: archsupport.texinfo,v $ $Revision: 1.15 $ $State: Exp $
;;
((name    "Targeting")
```

```
(struct (padding                        576) ;; Really point sets
        (fire_permission                int32)
        (vtarg_fire_technique           int32)
        (vassess_mode                   int32)
        (range                          float32)
        (fire_at_pos                    float32 2)
        (num_point_sets                 int32)
        (fire_type                      int32)
        )
(editor ("Fire Permission" CHOOSE_ONE fire_permission SHOW
                           ("Hold Fire" 1)
                           ("Fire At Will" 2)
                           ("Weapons Tight" 3))
        ("Fire Technique" CHOOSE_ONE vtarg_fire_technique SHOW
                           ("Simultaneous" 1)
                           ("Alternating" 2))
        ("Assessment Mode" CHOOSE_ONE vassess_mode SHOW
                           ("Closest To Self" 1)
                           ("Closest To Location" 3))
        ("Fire Type" CHOOSE_ONE fire_type SHOW
                           ("Distributed" VASSESS_DISTRIBUTED_FIRE)
                           ("Volley" VASSESS_VOLLEY_FIRE)
                           ("None" VASSESS_NONE))
        ("Range" DISTANCE range)
        ("Fire At Position" PLACE fire_at_pos)
        )
(initial (fire_permission CONSTANT 3)
         (vtarg_fire_technique CONSTANT 2)
         (vassess_mode CONSTANT 1)
         (range CONSTANT 2000.0)
         (fire_at_pos CONSTANT 0 0)
         (num_point_sets CONSTANT 1)
         (fire_type CONSTANT VASSESS_DISTRIBUTED_FIRE)
         )
(render REVERT)
)
```

The **struct** section specifies the data fields where the editable object information is stored. The **editor** section specifies which type of editor (i.e., choose-one menu, a text box, a meter) appears for each editable object. The **initial** section specifies the values of the editable objects when the ModSAF application first starts. For more information about the editor architecture (see section "The Editor Architecture" in *The Editor Architecture*).

3.1.4.4 Databases

Parameter files are also useful for specifying tables of data. Some examples of table parameter files are `echelondb.rdr` (describes the organization, of platoons, companies, sections, etc.), `physdb.rdr` (describes some of the physical characteristics of entities), and `constants.rdr` (a huge database that provides a gateway between C constant definitions and parameter files).

3.1.4.5 Related Libraries

The ModSAF libraries that implement parameter file functionality are `libotmatch`, `libparmedit`, `libparmgr`, `libreader`, and `librdrconst`. They are briefly described here. For more detailed information, refer to the Programmer's Reference Manuals for the respective libraries.

- `libotmatch` provides a uniform interface to databases keyed by SIMNET object types. This library does not provide any such databases itself, but rather supports a query function that will find the data associated with an object, or the data associated with a similar object if the specific query cannot be fulfilled. The interface is streamlined to support libreader style databases, although other database formats could also be used with a little effort.
- `libparmedit` implements the SAF parameters editor. Each vehicle subclass may register an editor which is used to modify its system-level parameters (those managed by `libparmgr`). These editors are then automatically integrated into the GUI.
- `libparmgr` provides a mechanism for changing parametric data at run time.
- `libreader` provides a facility for reading data files into convenient C structures. The syntax allows integers (in decimal, hexadecimal, or octal), floating point numbers, and strings to be mixed together using a structure reminiscent of lisp.
- `librdrconst` provides a gateway between C constant definitions and `libreader` files.

3.2 Physical Models

This section describes the physical models of ModSAF and how they interact with the other functional components. It will first cover the main functional components that make up the majority of the entities that ModSAF creates. It then covers the vehicle movement management routines. Next, it will describe how the DIS and PO protocols are used by these components to make the vehicle move about for other simulation nodes on the network, and how the vehicle information that is received off the network is used. We then describe the vehicle classes, especially the dynamics of the vehicles and how they work with the components library, the hulls, etc. We also provide a brief

description of some of the other physical models, how they interface with the rest of ModSAF, and how ModSAF uses the terrain database.

3.2.1 Components

A components class defines a common set of functions that are invoked on instances of that class, and the semantics of those functions. Other than defining these functional semantics, components classes don't actually *do* anything. ModSAF uses components classes to provide a level of abstraction away from specific component interfaces.

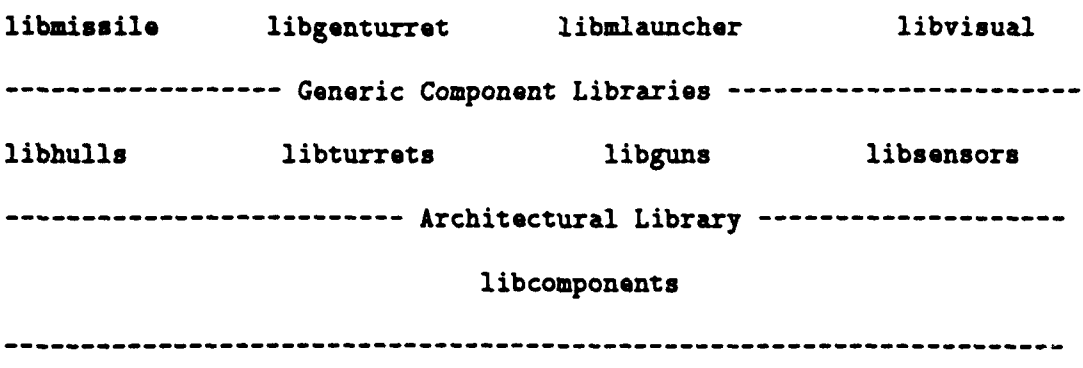
For example, although a vehicle may have one of several hull models, the interfaces to these models are all basically identical. `libcomponents` allows an application to give commands to its "hull" without knowing which hull model is being used.

The layering of the software looks like this:

```
---------------------------- Specific Layer ----------------------

TrackedHull FWAHull                 BallisticGun
WheeledHull RWAHull                 RocketLauncher    Radar
DIHull  MissileHull  GenericTurret  MissileLauncher   Visual

----------------------------- Generic Layer ----------------------

     Hull              Turret           Gun          Sensor

-------------------------- Architectural Layer --------------------

                              Components

-------------------------------------------------------------------
```

The software layering diagram shown above has been currently implemented via the ModSAF library structure shown below.

```
----------------- Specific Component Libraries ----------------------

libtracked
libwheeled
libdi
librwa
libfwa                              libbalgun            libradar
```



Many of these different generic components are tied to each other in various ways. Some, like **libguns**, has to have the sensor that it will use to track a specific target specified for it. Also, the specific physical specification of how these components mount on each other and function is specified in the reader file for **libphysdb**.

3.2.1.1 Hulls

Hulls is the ModSAF component class that defines which specific hull model the entity will use to move about. A single entity can only have one instance of a Hulls model.

Hull functions are accessed through macros defined by **libhulls**. These macros invoke **cmpnt_invoke** with a code number which identifies the function to run. **libcomponents** then runs this function for the particular hull mode via a jump table. This is all that the hulls interface does. If you look at libhulls, you will find that there is no real code there, just the macros.

When the ModSAF application gets set up to run, the **libhulls** initialization process directs **libcomponents** to define a hull component class. This information enables **libcomponents** to define a structure to accommodate all hull instantiations that a simulated object is allowed to have. The **libhulls** initialization process also tells **libcomponents** the number of its defined hull interface functions. This enables a simulated object's user data to be allocated enough space to hold the address of each of the hull interface functions defined in **libhulls**.

Since an application will interface to **libtracked**, **libfwa**, **librwa**, **libwheeled**, **libdi**, or **libmissile** through **libhulls**, a tank's movement control commands (performed by **libtracked**) and an airplane's movement commands (performed by **libfwa**) are both issued via the interface defined by **libhulls**. A command to change the controls is therefore the same whether the hulls component belongs to a tank or an airplane. What is different are the actual values used to set the

controls and those values are passed as arguments to the function. Similarly, an application can obtain information about the state of its hull through the `libhulls` interface.

There are a variety of hulls models available. For use and implementation details, or specific information on how each works, see the specific library documentation.

'fwa' **libfwa** provides a low fidelity model of a fixed wing aircraft's dynamics. Capabilities are modeled only to the second order.

'rwa' **librwa** provides a low fidelity model of a rotary wing vehicle's dynamics. It allows a vehicle to hover, turn in any direction while it flies at low speeds, and climb vertically. At higher speeds, it begins to behave somewhat like a fixed wing aircraft.

'missile' **libmissile** provides a low-fidelity model of missile vehicle kinetics. It also provides various target pursuit algorithms (pure-pursuit, lead- pursuit, lead-collision) appropriate for different types of missiles.

'tracked' **libtracked** provides a low-fidelity model of tracked vehicle dynamics. Capabilities are modeled only to the second order (maximum velocity, maximum acceleration), and they depend upon the soil type currently under the vehicle.

'wheeled' **libwheeled** provides a low-fidelity model of wheeled vehicle dynamics. As with libtracked, Capabilities are modeled to the second order, and dynamics depend on the soil type the vehicle currently rest on.

'di' **libdi** provides a model of dynamics for humans, or Dismounted Infantry.

3.2.1.2 Turrets

Turret functions are accessed through macros defined by **libturrets** Unlike with other component classes, there is currently only one instantiation of a turret model, **libgenturret**. **libgenturret** supports up to four instantiations per vehicle (i.e., a vehicle can have up to four generic turrets).

libgenturret implements an instance of the Turret class of components. It provides a low-fidelity model of generic turret dynamics and capabilities. Turrets that can not support 360 degree slewing (as reported in **libphysdb**) are supported. Turrets can be described as having either a continuous range of slew rates or set discrete rates.

3.2.1.3 Guns

Gun functions are accessed through macros defined by `libguns`. `libguns` currently acts as the generic component interface for two different models: `libbalgun` and `libmlauncher`.

'balgun' `libbalgun` (Ballistic Gun) provides a low-fidelity model of generic ballistic gun behavior which is suitable for ModSAF tank main guns and machine guns. libbalgun guns support burst shooting, multiple types of munitions, magazine loading, and table-driven hit probabilities. The capability to hit unintended targets is also supported. `libbalgun` uses a specific sensor, specified in its parameter file, to track and fire at a specific target.

'mlauncher'

`libmlauncher` provides a low-fidelity model of generic missile launcher behavior which is suitable for ModSAF ground vehicles and air vehicles. This library supports launching of multiple types of missiles. `libmlauncher` uses a specific sensor, specified in its parameter file, to track and fire at a specific target.

3.2.1.4 Sensors

Sensor functions are accessed through macros defined by `libsensors`. This library currently acts as the generic component interface for two different sensor models: `libvisual` and `libradar`. Sensor models can only tell you what they are currently looking at, not what they have seen in the past. Any sort of crew memory capability is handled at higher levels.

'radar' `libradar` provides an implementation of a vehicle radar model. Each instance of a radar model is specified as being attached to a particular physical component of the vehicle as specified in the physdb.rdr file, or the "hull". The radar model handles the modeling of line-of-sight, aspect- dependent radar cross-section and ground clutter. There can be up to eight radar sensors specified with any instance of an entity.

'visual' `libvisual` provides an implementation of an AMSAA visual detection model. The visual sensors are specified as being attached to a particular physical component of the vehicle as specified in the physdb.rdr file, or the "hull". There are two types of visual sensors available: optical contrast (normal vision) and infra-red. There can be up to eight visual sensors specified with any instance of an entity.

3.2.2 Vehicle Movement Management

There are a variety of movement management utilities available for use by tasks in ModSAF. The libraries **libmovemap** and **liblocalmap** provide near-term navigation of ground vehicles. The library **libcollision** provides a 3-D model of collision detection. The library **libroute** provides code for working with routes. The following sections describe in more detail the various parts of movement management in ModSAF.

3.2.2.1 Route Planning and Obstacle Avoidance

libmovemap supports the near-term navigation of ModSAF ground vehicles. Higher software layers provide the near-term goals and various movement constraints as input, and **libmovemap** generates near-term plans to achieve those goals. A variety of movement goals can be achieved, such as cross country route following, road following, cross- country station keeping within a unit, and road formation keeping within a unit. **libmovemap** will, to the best of it's ability, plan a path for the vehicle to accomplish its near-term goal, avoiding obstacles and, within as much as possible, the constraints placed on movement by the calling task. When unable to achieve the goal(s) outlined by the higher level software, **libmovemap** will indicate that it is "stuck", and it is the responsibility of the higher level routines to redefine the goals given to **libmovemap**.

libmovemap obstacles can be solid obstacles such as terrain features (buildings, trees), moving obstacles (other vehicles), and terrain areas that are treated as if they were solid obstacles (rivers). **libmovemap** constraints can also be non-solid, such as the edges of a road when road following. This can be thought of as a fog to be avoided, which gets thicker the farther away from the road a vehicle gets. With this internal representation, **libmovemap**can handle problems like moving off the road to move around another vehicle stopped in the road. The nature of this constraint keeps the vehicle on the road when it can, but allows **libmovemap**to move the vehicle off the road when it needs to. Road following, uncertainty of the future position of moving obstacles, and both types of station keeping use this methodology.

The principle behind **libmovemap's** algorithms is to combine all goals and constraints into a single internal representation (the "map"), then generate movement plans based upon that internal map. This map is three dimensional, the first two dimensions being spatial, and the third dimension being time. **libmovemap** will thus represent its plan as a set of points in the (X,Y) plane where it wants/expects the vehicle to be at particular points in time into the future.

The near term plans generated by **libmovemap**are stored by **liblocalmap**. This library is a subclass which provides per-vehicle management of movement maps created by **libmovemap**.

liblocalmap creates and initialize the movement map when a vehicle is created, and destroys it when the vehicle is destroyed. The library provides a function to get the current movement map.

3.2.2.2 Collision

libcollision provides a 3-D physical model of collision detection. It can detect collisions with other network entities (platforms, missiles, and structures) as well as treelines, buildings, and the ground. This library is also responsible for generating and processing collision PDUs. **libcollision** uses a parametrically controlled timing heuristic to filter out redundant collisions (such as when both parties in the collision send each other collision PDUs).

libcollision handles the simple detection of intersections with nearby features used for slow moving ground vehicles, as well as the more complex detection of collisions needed by a fast moving vehicle which may jump a considerable distance between ticks. For example, a missile traveling at Mach 1 and ticking at 2 Hz will jump about 165 meters each tick. Hence, a ray must be run from the old position to the new position to determine if any features were intersected along the way.

3.2.2.3 Route Manipulation

libroute provides a generic system for manipulating routes. Routes are stored as lists of route sections, where each section can either be a list of points or a road. Roads are also lists of points, but contain additional information to help create LineClass POs. These routes are used by the various tasks to accomplish moving around the terrain. **libvmove** passes routes to **libmovemap**, which uses the general route to create its planned route. **libvflwrte** uses the route directly to fly a fixed wing aircraft.

3.2.3 On the Network

One of the main functions of ModSAF is to create entities on the network for other simulations and simulators to interact with. This is accomplished through the use of the Distributed Interactive Simulation (DIS) database. At the same time, ModSAF needs to communicate between its own SAFsims and maintain the state of its own simulation. This is accomplished through the use of the Persistent Object (PO) Protocol.

3.2.3.1 DIS database

The DIS database is a simulation's window on the state of a network exercise. We define an exercise as meaning any simulation(s) and/or simulators that are running on the same terrain database and communicating with each other about the state of the world.

The DIS database uses standardized network protocols such as the SIMNET and the DIS protocols to create, update, and delete entities in the network DIS database. A network entity is something that has an existence over time, such as a vehicle or missile. These entities can move about and report their changes in location, orientation, appearance, and state. The DIS protocols allow any simulation that joins in the exercise to quickly find out what the current state of the world is. The DIS network database also includes temporary things, such as events that do not need to have an existence across time (explosions) and environmental effects such as the radar emissions put out by a vehicle and the state of the weather.

ModSAF supports two types of DIS simulation protocols: the SIMNET protocol, developed under the SIMNET program, and the IEEE DIS protocol.

3.2.3.2 PO Database

The Persistent Object (PO) database communicates information between all running ModSAF SAFsims and SAFstations. It can be thought of as a carrier for any object that has a state and that wants to let any other SAFSim and SAFStation know about it. Information created by one is available for all, and can be modified elsewhere. Like the DIS database, the PO database mainly deals with creation, changes of state, and deletion of these objects.

Anything that has a state can share information. Lines drawn on the GUI are a set of (X,Y) locations with a drawing style. Vehicles are a combination of the DIS database information about them once they are simulated, and the PO information used to create them. What a vehicle is doing, and the commands it has been given, are also shared with the PO database. As the state of objects change, packets are sent out updating the state of the other PO databases on the network.

3.2.3.3 Dead Reckoning

Dead reckoning is a vehicle movement method that reduces DIS packet traffic, thus increasing the number of DIS network entities that can be simulated in a single exercise. In any simulation where moving vehicles interact, there is a need for up-to-date information on where the vehicle you are

interacting with is located. For instance, manned simulators need current information at the visual system update rate so that the graphic images move smoothly. This update rate is historically on the order of 15 times per second. In order for this information to be available, the other simulations and simulators on the network would have to retransmit their position, orientation, etc., at that same frequency.

Dead reckoning takes into consideration the fact that entities in the real world do not move from moment to moment in a totally random fashion. If a vehicle is moving in one direction with a particular speed, a few milliseconds later it will still be moving in approximately the same direction at approximately the same speed. Any alteration of this movement will take time to occur, such as when a tank turns at a sharp curve in the road.

Based on this observation, dead reckoning assumes that if a vehicle tells the other DIS simulations where it is and where it is going, they can then tell where it will be in the short term without that vehicle having to tell them again. If the vehicle does start to deviate substantially from where the other DIS simulations expect it to be, then it will tell them where it actually is and where it is now going. This is called Remote Vehicle Approximation (RVA).

Under the SIMNET protocol, there was only one simple RVA algorithm in which the current vehicle position would be extended linearly on its last direction of travel at a constant speed. If the simulation that is responsible for creating the entity finds that the orientation or position of the vehicle is off by more than a predetermined threshold, it sends another update.

The DIS protocol can utilize even more complex algorithms that take into account acceleration and other factors to even further reduce network traffic, while still using the concept of thresholding and retransmission.

3.2.3.4 Creating Network Entities

libcreate uses the PO protocol to provide the following services for ModSAF:

- Distributed creation of simulated entities with load balancing between computers of the same **simulatorType** (such as simulator_I.I._SAFSIM, a type that indicates the computer's ability to simulate entities) as directed by the PO database information posted on the network.
- Application-directed hand off of simulated entities from one simulator to another.
- Fault-tolerant takeover of simulated ModSAF entities belonging to a simulator of a similar **simulatorType** when that simulator crashes or exits.

`libcreate` will tell `libsafobj` to simulate an entity if it determines that it is appropriate for this simhost to do so.

3.2.3.5 Remote Entities

`libremote` acts as a handler for all non-locally simulated entities on the network. It receives packets for remote entities, and passes them on to the appropriate subclass for the type of packet. If the packet received is for an entity that the simulation does not know about yet, it will cause an entity of the appropriate type to be created, and then pass the packet to the new object's subclass for handling.

3.2.3.6 SAF Object Classes

All network entities in ModSAF are stored as an instance of the class safobj. Remote entities are represented by a few standard subclasses to allow them to be tracked by the other systems in ModSAF, such as the GUI, and to allow interaction with them by the ModSAF entities themselves. Local entities will have whatever subclasses that are specified for them in the data files used at time of creation. Many of these subclasses will interact directly with the network.

3.2.3.7 Entity Management

`libentity` provides a uniform interface to all network entities represented within ModSAF. It handles the information coming into the simulation from out on the network. All DIS database state PDUs are generated and given to `libpktvalve` for transmission by `libentity`. `libentity` is also responsible for running the RVA algorithms, both for sending packets for locally simulated entities, and for estimating the current location for remote entities.

`libentity` provides a collection of 'get' and 'set' functions for each entity state variable. These functions act as a lazy evaluation buffer, which prevents conversions to or from network representation until absolutely necessary, and then saves those converted values until they once again become out of date.

3.2.3.8 ModSAF I/O Port

The ModSAF system is defined in layers. The I/O port layers are:

```
Application level layers (libvtab / libsafobj...)
-------------------------------------------------

--> libpktvalve layer <--
-------------------------------------------------

Network level layers (libassoc / libnetif)
```

The bottom layers handle packets of various protocols, and they are fairly context free (for example, **libassoc** doesn't know anything about the simulation protocol). The top layers are context rich, and they do all the decision making and cause all the environmental influences (move vehicles, shoot weapons, etc.).

The layer which bridges the top and bottom layers contains the library **libpktvalve**. This library provides a sort of bi-directional valve. In this model, application libraries which need to know about the arrival of specific types of packets register handlers for those packet types (protocol/kind pair, such as Simulation/Impact, or Data Collection/Status). **libpktvalve** passes (via these handlers) these packets to all interested parties. The library receiving the packet processes the packet in the most appropriate manner (i.e., some packets will take effect immediately, others will be queued for a particular vehicle or group of vehicles).

In addition, libraries can specify filters to prevent unnecessary packet handling within **libpktvalve**. When a packet that has been requested by any module is received from the network, its protocol and kind are checked to determine if any filters have also been registered. If so, all such filters are checked, and at least one must pass before the packet will be accepted. This is done before the packet is copied out of system memory into application memory (on machines where such functionality is available).

In the outbound direction, application libraries pass all outgoing packets through this layer, where they may be looped back to other libraries if they are wanted. Some packets, such as impacts, are processed on loopback; others, such as vehicle appearances, typically are not. When they request packets of a particular type, application libraries also specify whether they want them only when generated locally, remotely, or either. **libpktvalve** also automatically filters packet types which have not been requested by any libraries.

3.2.4 Vehicle Classes

libsafobj is the parent class of SAF objects. It manages the SAF objects and provides the class superstructure in which all vehicle subclasses reside. There are two kinds of SAF objects: locals and remote. The subclasses which make up locals, and the default configurations of these

subclasses for each type of vehicle, are stored in the configuration files for that vehicle. Remotes are always made up of a fixed set of subclasses (entity, pbtab, and others).

Examples of ModSAF local vehicles include airplanes, tanks, and missiles. Since the simulation requirements are less demanding for a missile, its class superstructure need not be as extensive as that needed for an airplane or tank.

There are a variety of vehicle subclasses. A subclass is something that needs to 'hang' data on an object, and it also defines the allowable operations that can occur on that data. For instance, **libmovemap** is a utility that performs local planning, and it uses the **liblocalmap** subclass to store its data for it. **liblocalmap** is a simple class since it allows only a few operations on itself, mainly information retrieval and storage.

The following is a list of some of the vehicle classes:

'detonation'

libdetonate provides a model of proximity detonation. It can detect detonations due to proximity with other network entities (platforms, missiles, and structures). Proximity detonation due to the ground, buildings, and other terrain features is not yet supported. This library will generate impact PDUs if told to do so for a given vehicle.

libdetonation determines that a detonation should occur if the vehicle (usually a missile) has passed closer to the target vehicle over the previous tick than the 'detonation_radius' specified for the model instance. This will occur when the missile is no longer getting closer to the target (i.e., has passed it). If the local minimum has been passed and the distance to the target is greater than the specified 'detonation_radius', then a near miss is declared.

There are two models for selecting potential target entities: low-fidelity and high-fidelity. When low-fidelity detonation is used, a list of potential targets must be supplied by the simulation. These are the ONLY vehicles that will trigger detonation. When high-fidelity is used, **libdetonate** builds a suitable list of nearby vehicles to check. This model is considerably more expensive.

'designator'

libdesignator uses the laser designation PDU to designate targets for missiles and the marking of enemy vehicles.

'fcs'

libfcs (Fire Control System) provides an abstraction for ModSAF vehicles that have a large number of **libguns** components, such as aircraft. **libfcs** provides an interface where weapon launcher commands can be specified in terms of the munition which will

be fired. This library uses selection algorithms to determine which launcher is most suitable to direct the `libguns` commands for a given munition.

The name of each component to be controlled by `libfcs` must be entered in the parameter file for the vehicle. Each name will be the name of the gun component as specified in the Components listing and the vehicle phydb.rdr entry.

'`dfdam`'

`libdfdam` implements damage assessment from direct fire weapons. It is responsible for registering `libpktvalve` (see section "Overview" in *LibPktValve Programmer's Manual*) handlers to listen for and store Impact PDUs for later processing by the vehicle. When the `libdfdam`subclass processes one of these PDUs, table lookup from a direct fire damage table will assess a result:

- Catastrophic Kill
- Firepower and Mobility Kill
- Firepower Kill
- Mobility Kill
- No Damage

Once damage is calculated, callback routines registered with the library at initialization are begun. These callbacks correspond to the different results an application will have when damage is assessed, such as changing a vehicle's appearance (`libentity`) or modifying other vehicle subclasses to transition to particular damaged states. If the resulting damage has already occurred, then no additional damage occurs.

'`ifdam`'

`libifdam` implements damage assessment from indirect fire weapons. It is responsible for registering `libpktvalve` (see section "Overview" in *LibPktValve Programmer's Manual*) handlers to listen for and store Impact PDUs for later processing by the vehicle. When the `libifdam`subclass processes one of these PDUs, table lookup from an indirect fire damage table will assess a result:

- Catastrophic Kill
- Firepower and Mobility Kill
- Firepower Kill
- Mobility Kill
- No Damage

Once damage is calculated, callback routines registered with the library at initialization are begun. These callbacks correspond to the different results an application will have when damage is assessed, such as changing a vehicle's appearance (`libentity`) and modifying other vehicle subclasses to transition to particular damaged states. If the resulting damage has already occurred, then no additional damage occurs.

'localmap'

liblocalmap is a subclass which provides per-vehicle management of movement maps created by **libmovemap**. The library creates and initializes the movement map when a vehicle is created, destroys it when the vehicle is destroyed, and provides a function to get the current movement map.

'collision'

libcollision provides a 3-D physical model of collision detection. It can detect collisions with other network entities (platforms, missiles, and structures), as well as treelines, buildings, and the ground. This library is also responsible for generating and processing collision PDU's. The library uses a parametrically controlled timing heuristic to filter out redundant collisions (such as when both parties in the collision send each other collision PDU's).

'entity'

libentity provides a uniform interface to all network entities represented within SAF. In addition to bookkeeping functions (create, destroy, activate, etc.), **libentity**provides a collection of 'get' and 'set' functions for each of the entity state variables. These functions act as a lazy evaluation buffer, which prevents conversions to or from network representation until absolutely necessary, and then saves those converted values until they once again become out of date.

The entity subclass of the vehicle is also responsible for maintaining that vehicle's location in the position-based table. For local vehicles, this update occurs whenever the position is changed. For remote vehicles, this update occurs either (1) when a packet is received which modifies the remote vehicle's position, or (2) when the RVA-derived position of the vehicle exceeds a tolerable error threshold from the position represented in the table.

'genradio'

libgenradio provides rudimentary radio communications to ModSAF entities. It allows transmission of ASCII strings using the radio protocols of SIMNET and DIS, including issuance of transmitter and signal PDUs.

'supplies'

libsupplies provides a simple facility for supply management within a vehicle simulation. The vehicle subclass maintains a list of munition types and quantities. A negative amount signifies that the amount is never decremented. A single type of munition may be stored in more than one place. Functions are provided to check the levels of a given supply, to set the levels explicitly, or to decrement the amount.

'pbtab'

libpbtab supplies a position based vehicle table for optimization purposes. This table is established with a grid of squares overlaid on the terrain database. Each entity in the simulation is stored in the list associated with the grid square it currently lies within.

ModSAF entities can find vehicles they are close to by accessing the local grid square lists in the table. This is far more efficient than accessing the entire list of vehicles in the simulation and having to determine which are closest.

libpbtab is dependent upon each entity to keep its own entry in the table up to date.

'physdb' **libphysdb** provides a database of physical vehicle information. The database is accessed using the **libotmatch** function **otm_query** (see section "otm_query" in *LibOT-Match Programmer's Manual*).

The types of information stored in the physical database are as follows:

Data which does not vary between instantiations of an object type.

Hence, the number of turrets on an object is appropriate, but the type of ammunition fired from the guns is not.

Data which is needed to interact with or simulate an object.

This excludes things like the icon used to draw the object on a PVD, and includes things like the size of an object.

Data which is needed to interoperate correctly with CIGs.

For example, the **model_base_adjustment** parameter which distinguishes the bottom of an object from the center of the object (traditionally, this has only been used for air vehicles).

Much of the information used with the Vehicle Components is specified in the physdb.rdr file. These physical definitions are then used with the strings they are defined with to tie component parts to the vehicle. Many components are also cross linked for control this way.

'Safsoar'

libsafsoar is the primary interface between the SOAR system and ModSAF aircraft simulation. For more information, see the "Interface Control Document for ModSAF/SOAR".

'preview'

libpreview provides a complete interface to **libxcig**. It defines a model database (similar to a DED on a real CIG) which translates SIMNET object types to 3-D models. It also updates the display with the locations of nearby vehicles, and provides simple "stealth" controls (including free fly, ground hug, and attached modes). It is a vehicle subclass, so each vehicle will update its position in the **libxcig** model.

'pvd'

libpvd provides the user with tools to control the Plan View Display, and it graphically displays network traffic such as vehicles and weapons fire. It builds these controls into the **libsafgui** environment created by the application.

libpvd also defines the pictures used to draw vehicles of different types. This information is stored in a **libotmatch** style database called 'pvd.rdr', using the picture

language defined by `libtactmap`. As the comments in that file explain, the software defines three GCs (black, white, and team-colored) and three coordinate systems (compass, hull, and turret). The software sets these correctly for each vehicle when it is updated on the map.

The library uses `libtactmap` to draw vehicles, either as pictures or as `libbgr` icons controlled by the "Show As" menu bar item. Scale editables are provided in the Defaults editor for each PVD to control the scale of the pictures or icons.

`libpvd` is a vehicle subclass so the `libtactmap` update can be the responsibility of each vehicle.

'`stealth`' `libstealth` provides `libentity`-like functionality for interacting with stealth views. It is a vehicle subclass so that the remote stealth vehicles will have the information available to do stealth control commands from the ModSAF user interface, and so the stealth can also be displayed. In this model, stealths (both remote stealths and local stealth- preview views) are placed in the vehicle table and given a 'VTAB_REMOTE_STEALTH' or 'VTAB_LOCAL_STEALTH' vehicle type. Applications can interact with all stealths the same way, regardless of their type. In the future, this library may be extended to support different network stealth protocols as well.

The stealth (also referred to as the Flying Carpet) presents a three- dimensional view of the simulated battlefield. Data packets projected onto the appropriate network exercise allow the objects they represent to be viewed on the stealth. Much of the `libstealth` code deals with stealth protocol packets.

3.2.5 Vehicle Class Tools

There is currently only one Vehicle Class tool library, `libaccess`.

'`access`' `libaccess` provides a convenient interface to fetch a collection of state variables from a single vehicle. Each vehicle subclass library (i.e., `libentity` or `libvspotter`) defines a key for each variable which can be fetched this way, and applications can then fetch an arbitrary number of these value with one access_get call to libaccess. Once these keys are defined, one access_get function with multiple key arguments can replace multiple accessing functions.

3.3 Behavioral Models

3.3.1 Tasks

The foundation of the ModSAF command and control framework is the idea of a *task*. A task is a behavior performed by an individual on the battlefield. Tasks may be done on behalf of a unit (company road march) or they may directly control a physical system (drive toward a waypoint). Tasks may be done to achieve a mission objective (attack an objective), or they may be done continuously, independent of the mission (scan for enemy). Tasks may be representations of actual battlefield behavior (run for cover).

A task is represented in the Persistent Object database in three parts:

1. The task *model*. The software representation of a task is a finite state machine which has several task specific states, and always has the **ended** and **suspended** states. The state machine implementation and the data structures it operates on are together referred to as a task *model*.
2. The task *parameters*. The parameters express different options available in executing the task. They control the execution of the task in the context of a single mission. For example, the route to follow and the objective to attack are task parameters.
3. The task *state*. The state is used by the task model during execution. This public version of the task state will hold information which needs to be shared by the user interface for data analysis or fault tolerance needs.

The following sections give a more detailed description about the task architecture:

3.3.1.1 Task Data Structure

A task model interacts with four separate bodies of data, as described in the sections that follow. Some are *shared*, meaning they exist in the Persistent Object database, where any application program on the network can examine them. Some are *local*, meaning they are only available in the simulation process which is simulating the vehicle that is executing the task. The format of this data can be either *public*, meaning the structures are defined in public headers which other software libraries can use, or the format can be *private*, meaning the structures are known only to the model. The four bodies of data are as follows:

System Parameters (Shared, Public)

System parameters control how a task does its job. These parameters are set for each kind of platform (F-14 vs. M-1, etc.) to specifically guide execution of the general task in a manner appropriate for that platform. These parameters are also used to tune execution, or to allow system-level field modification without the need for programming

or compilation. System parameters originate within data files, but can be modified at run time via the Persistent Object database.

Task Parameters (Shared, Public)

Task parameters control the execution of the task in a single mission. These include mission parameters such as the route to follow or the rules of engagement.

Shared Task State (Shared, Private)

Shared task state is maintained by an executing task. It includes the primary state machine state variable, as well as more specific state information (i.e., the current route point, or a list of detected targets) which needs to be shared by the user interface for data analysis or fault tolerance needs. This state must not change frequently, since every change leads to additional network traffic.

Local Task State (Local, Private)

Local task state is also maintained by an executing task. It will generally contain either more specific versions of the data in the shared task state, or more efficient internal representations of the same information. Local task state may change frequently without any performance detriment. However, this information will not be available for monitoring, analysis, or fault tolerance needs.

Enough state information must be shared so that it is possible for the unit leader who assigned the task can monitor the task progress. Furthermore, enough state information must be public so that another individual or unit leader can take over execution of the task with no observable difference. This takeover may be required when a machine fails or when a vehicle is destroyed in battle. The programmer needs to encode the task in such a way that the public state does not change too frequently because of the bandwidth limits in the communications media.

A task is described in the Persistent Object (PO) protocol by an object of the class Task, and another object of the class TaskState (shown below, with padding omitted):

```
type TaskClass sequence {
    model       UnsignedInteger (32),
    frame       ObjectID,
    state       ObjectID,
    suspended   Boolean,
    stateReuse  Boolean,
    refcount    UnsignedInteger (8),
    size        UnsignedInteger (16),
    data        array (size) of UnsignedInteger (8)
}

type TaskStateClass sequence {
    refcount    UnsignedInteger (8),
```

```
    size        UnsignedInteger (16),
    data        array (size) of UnsignedInteger (8)

}
```

The task object holds the parameters of the task, while the state object holds the run-time state.

Whenever a task object is created, it contains a task model (a reference to the body of code and data which executes the task in the simulation), the frame in which the task resides (see Section 3.3.1.3 [Task Frames], page 40, the state object which it uses, and task data. Any attribute of a task can be changed after initial creation (although the size is generally a constant). The task data holds the task parameters and the task state data holds the task state. These are interpreted by the task state machine which implements the task.

Initialization of the parameters and state of a task is the responsibility of the unit leader who assigns the task. Since the format of the state is private, the body of software which initializes state resides within the task library, and is invoked by the unit leader. An executing task updates the shared state whenever appropriate.

For example, the shared representation of a tank's state, while it is running the spotter task which scans for objects, includes a list of detected objects and their presumed alignments. This state is updated periodically at a rate specified in the system parameters of the spotter model. Also included in the system parameters for the tank's spotter task is a list of sensors that need to be read and manipulated. If other tasks within the tank need a list of the tank's spotted vehicles, those tasks can call the `libspotter` function `spotter_get_spotted`, which will return the local representation as a list. Other applications, such as the user interface, can probe the shared representation of the tank's spotter task by passing the entire task to the `libspotter` function `spotter_get_spotted_from_state`.

3.3.1.2 Augmented Asynchronous Finite State Machine

All behavioral tasks have been implemented using *asynchronous augmented finite state machines* (AAFSM). We call them *augmented* state machines, because they can influence and use many variables other than their state variables. They are *asynchronous* because they may generate outputs in response to events in the simulated environment.

Our state machines are defined as modified state transition tables because this allows easier interpretation and debugging. A preprocessor utility, '`fsm2ch`', reads the AAFSM format and

translates it to C code for compiling. The preprocessor can also generate transition diagrams as fig source, which can be reviewed using xfig and printed using fig2dev.

The '**fsm2ch**' utility also has limited code-generation facilities to simplify writing tasks which control other tasks (unit behaviors), and tasks which generate task frames (reactions).

For a detailed description about the AAFSM code generator, refer to the (see section "AAFSM Code Generator" in *LibTask Programmer's Manual*).

3.3.1.3 Task Frames

Task frames are used to group a collection of related tasks that run at the same time. The task frame that represents a certain phase of a mission can consist of many tasks. For example, a road march task frame consists of the following tasks: unit traveling, actions on contact, and enemy detection. Many tasks have parameters in data files, and some tasks also have parameters defined by the user. For example, the collision task has parameters in a data file that define what obstacles to avoid (trees, buildings, ground, platforms, missiles). The occupy position task has the following parameters defined by the user: battle position, left target reference point, right target reference point, and engagement area. Only some of these parameters may be edited by the operator, while others are system-level parameters that the user may not change. A task frame can customize the operation of the tasks which it uses by defining certain parameters in a data file.

A unit executes a stack of frames. As a mission proceeds, a unit will sometimes need to switch to a different task frame for awhile, then resume the original task frame. Similarly, the user will often want to override a preplanned behavior temporarily. Both of these needs can be addressed by *task frame stacks*. A vehicle will have a task frame for each of its roles. A role is a set of units that some vehicle is responsible for (such as a company commander or platoon leader). One of these roles is always that of the vehicle itself. New task frames may also be pushed on the stack as the result of a reactive task (such as reacting to the enemy), or at the request of the user (in response to a tactical emergency, or *TAC/E*). The purpose of the task frame stack is to have the ability to return to a supended mission.

The assignment of a task frame (or a sequence of task frames) to a unit is a two-step process. First, the task frame is marked with the unit ID, indicating who is to execute the frame. Then, when the change to the task frame is seen by the machine simulating the vehicle responsible for that unit, the task frame may be pushed onto the unit's task frame stack. The unit's task frame stack contains all of the task frames for the unit. This includes the unit task frame that is currently executing, and all of the suspended unit task frames.

As an optimization, each task frame in the stack is marked as either *opaque* or *transparent*. The tasks in each transparent frame at the top of the stack are merged with the tasks in the topmost opaque frame. This allows tasks to be layered over the original mission framework, without requiring that all unaffected tasks be duplicated into the topmost frame. The topmost frames are the frames that the unit is currently executing, and the rest of the frames are suspended. The use of transparent task frames simplifies the following user interface needs:

Pre-programmed Instructions

While ground missions can generally be defined via a series of isolated task frames executed in sequence (e.g., road march, pre-battle march, attack), air missions are more easily defined using a single task frame with occasional modifications. This latter method can be easily achieved by attaching *instructions* to geographic or temporal features. Each instruction changes the parameters of some subset of mission tasks. In fact, this method of mission construction turns out to be useful in the ground domain as well (e.g., "change formation when you cross this line").

These instructions can be easily implemented by placing the modified task into a transparent frame, which is placed at the top of the stack when the conditions for the instruction have been met.

Temporary Run Time Modifications

As a mission proceeds, the user will often want to temporarily alter the mission definition. When this alteration is merely to change a parameter of a currently executing task, a transparent frame can be placed at the top of the stack, containing only the impacted task. All other tasks will continue to operate normally. When the user wants to return to the original behavior, the transparent frame is removed.

The phase of a mission consists of a preparatory task frame and an actual task frame. The preparatory task frame sets up the mission phase for the actual task frame. An example of this is a mission phase that is supposed to be executing "on order". The unit needs a task frame to execute (i.e., Halt) while waiting for the order. The preparatory task frame is used for this. If the unit is running an "on order" Move (contact), the unit will execute a preparatory halt task frame until the order is given. When the order is given, a Move (contact) task frame will be executed.

Another use of preparatory task frames is for controlling the subtasks of a unit. When an entire unit is heading toward a control measure, the next task frame cannot be executed until all the units have reached the control measure. The first vehicles to reach the control measure must wait for the other vehicles. When a vehicle reaches the control measure, its preparatory task is executed and waits for all of the other vehicles to reach the control measure before beginning the next phase of the mission.

For example, there are three platoons in a company, and each is given a separate route to follow to a control measure. When the control measure is reached, two platoons will execute an occupy position, while the other platoon will assault the enemy. The platoon that has an actual assault task frame will perform a preparatory halt task and wait for the rest of the company to reach the control measure before executing the assault task frame. The two other platoons will execute a preparatory occupy position task frame when they reach the control measure, and will perform the actual occupy position task frame when the rest of the company reaches the control measure.

Although the ModSAF software currently pushes every received task frame, the decision to push the frame may be contingent upon battlefield uncertainties (did the message get through, does the commander of the unit choose to obey the order, etc.).

3.3.1.4 Task Manager

The *task manager* is the body of software (spread over a few libraries) responsible for the implementation of all the architectural framework algorithms. Each task model, at application startup, registers itself with the task manager. At this time it specifies the following things:

Model Number

This number is used by the task manager to map between a task in the persistent object database and the body of software that implements that task. Model numbers are protocol constants.

Entry Points

Each task gives the task manager the entry points (function pointers) for all task functions which the task manager may need to invoke:

`Params` This is a function which the task manager calls when the parameters of a task change. This frees the task implementor from having to monitor the PO database for such changes.

`Predicate`

This function is called for "enabling tasks" to determine if their conditions have been met. For example, an enabling task that monitors crossing lines will return `true` if the line has been crossed and `false` otherwise. Tasks that are not simply enabling tasks should pass a function which indicates if they are in their 'ended' state. This way, a subsequent task frame may refer to an executing task as its enabling task, to trigger a transition when the task completes.

`Start` This function is invoked when a task is first started, or is restarted after having been suspended (see **Suspend** below).

Tick This function is invoked once each vehicle tick when the task is running. It provides the task with a chance to update its internal state over time. Purely event-driven simulations might not use this entry point.

End This function is invoked when the task is forcibly ended, such as when the task frame in which it resides is popped off the task frame stack.

Suspend This function is invoked when a task that has not ended ceases to run temporarily. This generally happens when an opaque task frame is pushed onto a task frame stack.

Shared Data Size

Although the task manager does not need to know the specifics of each task's implementation, it does need to know the size of the parameters and state used by the task for network operations. The task manager simplifies task writing by automatically making updates to the PO database when tasks change their state.

Before and After Lists

The tasks executed by an individual may be interdependent. Each task may rely on input from other tasks (which make up its *before* list), and may provide output to other tasks (which make up its *after* list). For example, if a plane is assigned a vflwrte (vehicle follow route) task, one of the *before* tasks is vtakeoff (vehicle take off). This *before* task ensures that the plane is in the air before it begins to fly along the route. The task manager also ensures that the tasks are ticked in an order which minimizes latency in the information flow.

With this information provided by each task model, the task manager then uses the following algorithm each tick to determine which tasks to run:

1. Get a list of roles this vehicle is playing. A role is a set of units that some vehicle is responsible for (company commander, platoon leader, etc.). There is a task frame for each role that a vehicle is playing. One of these roles is always that of the vehicle itself.
2. For each role, determine if any of the current task frames have been unassigned. Remove those task frames from the stack.
3. For each role, determine if the enabling tasks of any subsequent frames indicate that those frames should be executed. If so, start one new frame. The frames immediately after the topmost frame on the task frame stack, and those immediately after the topmost *opaque* frame on the stack are tested. If more than one frame is enabled simultaneously, the user is notified and one frame is chosen through task priorities (see section "LibTaskPri" in *LibTaskPri Programmer's Manual*). Opaque frames are chosen before transparent frames.
4. For each role, traverse the current frames, including the background frame and all frames on the task frame stack down to the first opaque frame, and collect a list of tasks to execute.
5. Sort the execution list such that the **before** and **after** restraints of each task are not violated.

6. Compare the resulting list of tasks and determine if any tasks executed last tick are not in the new list. Of those, suspend those which are still running.
7. Execute the tasks in order.

In addition, the following algorithm is used when task frames are changed within the PO database:

1. If the task frame refers to a unit which is locally simulated, AND
2. If the task frame has a NULL previous mission pointer (meaning it is either the first frame of a mission, or it is a TAC/E assigned by the user), AND
3. If the unit chooses to execute the mission (a unit may choose not to execute the mission if a radio message is received to that effect), [we currently do not perform this test] THEN
4. Push this frame onto the top of the units task frame stack. Execution will begin next tick.

3.3.1.5 Task Priorities

Tasks register the criteria under which they need to control a critical resource (such as controlling a vehicle's movement). At each tick, these criteria are resolved against a set of priorities to determine which task gets that resource for that tick. For example, if both the Collision and Move tasks want to control movement, the task with the highest priority (hopefully Collision) will be allowed to control movement of that vehicle. **libtaskpri** provides the task management service to support mutual arbitration (see section "Arbitration" in *ModSAF Programmer's Guide*).

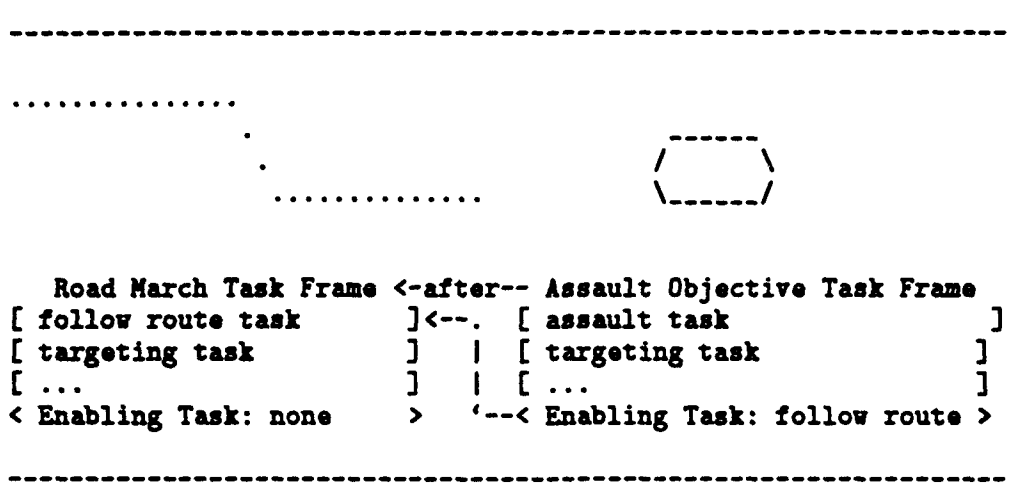
The AAFSM code generator (see section "AAFSM Code Generator" in *LibTask Programmer's Manual*) is aware of **libtaskpri**, and provides convenient notation for interfacing to this library. Thus, many tasks that use **libtaskpri** never explicitly call **libtaskpri** functions. They are called by generated code instead.

libtaskpri contains a list of tasks that compete for resources. The tasks that compete for movement are: VFlyGrndAvoid, VTakeOff, VCollide, VTargeter, VMove, VLand, VATAInt, VCAP, VFlwRte, and VOrbit. It is more important for a vehicle to avoid a collision than to continue moving along a route. The task VCollide is higher in the list than VMove, so VCollide will be allowed to control a vehicle's movement over VMove. **libtaskpri** manages all of these types of requests and determines which task will control each resource, according to the given priority list.

3.3.1.6 Enabling Tasks

Often the user can predict certain contingencies, and would like to specify alternate missions for those cases. The *enabling task* provides a simple way to express even the most complicated contingencies.

A *mission* is defined as a set of task frames which are linked together to form a sequence. In a simple mission, each task frame specifies one task in the previous task frame which must complete before the subsequent frame starts. For example:



In this example, the Assault Objective task frame (which follows the Road March task frame) refers back to the *follow route* task in the Road March frame as its enabling task. Thus when the follow route is finished, the Assault Objective frame will be enabled (pushed onto the stack in place of the Road March frame).

A more complicated case is when the execution of a subsequent task frame is not contingent on completing an executing task. It could be triggered by seeing an enemy unit, passing a point, time elapsing, etc. The architecture allows *predicate* functions like these to be defined as enabling tasks. These tasks are unusual in that they are only executed when the task frame which depends on them is *not* running.

For added flexibility, the architecture allows the results of enabling tasks to be combined in arbitrarily complex logical expressions. For example,

"attack if you spot the enemy AND you are healthy AND it is before 5:00."

"attack if you spot the enemy OR you cross phase line beta."

There are three types of enabling tasks: in phase, on order, and control measures. In phase occurs when platoons in a company wait for each other before they begin the next phase of their mission. If a unit is assigned a mission with an on order, this phase of the mission will not begin until the order is given by the user. When each platoon in a company reaches the control measure, they will wait for the rest of the company before beginning the next phase of the mission.

These enabling tasks are defined the same as other tasks , with a model number, parameters, and state.

3.3.1.7 Reactive Tasks

Reactive tasks are much like regular tasks, but with a few differences. The differences are in the design and thinking.

A reactive task monitors the current situation and then responds to certain changes in the situation. These changes are defined by the operator. The reactive task responds by pushing a new reactive task frame. Therefore, the reactive task that monitors the situations must define the reactive task frames to be pushed and figure out the parameters of the tasks in the desired task frames.

Reactive tasks are different from regular tasks in that they are a *sponsoringTask* for the reactive task frame. When a reactive task frame is pushed, the reactive task that pushed it is the *sponsoringTask*. The reactive task still remains active, even though all other tasks in the original task frame are suspended. This is done so the reactive task that caused the reactive task frame to be pushed can monitor and end the reaction at any time.

For example, the reactive task 'Actions on Contact' is running in the Move task frame. When an enemy is detected, this reactive task will react by spawning a task frame. One task frame that 'Actions on Contact' can spawn is the Occupy Position task frame. So, when the Occupy Position task frame is started, the previous Move taskframe is suspended. When the Move task frame is suspended, all tasks in the task frame are also suspended except the *sponsoringTask*, 'Actions on Contact'. This task will remain active to monitor the Occupy Position Taskframe. It can then stop the Occupy Position task frame when the enemy situation has changed.

3.3.1.8 Foreground and Background Tasks

Each unit has a background task frame in which it may place tasks that will continue to run after the current tasks have completed. Examples of background tasks are 'Avoiding Collisions' and 'Spotting Other Vehicles'. These types of tasks should always be running (in most situations).

A foreground task frame is the one that is currently executing. If no mission is running, then the foreground task does not exist. Only one foreground task frame may execute at a time. This frame contains the tasks for the current mission of the vehicle.

3.3.1.9 Unit and Vehicle Tasks

Vehicle level tasks usually involve lower level behaviors such as avoiding obstacles and targeting. Unit level tasks control higher level behaviors such as assaulting the enemy or occupying a position. The unit level tasks sometimes spawn the vehicle level tasks. For example, Utraveling "unit traveling" spawns Vmove "vehicle movement". Utraveling handles the 'big picture' of moving, while Vmove handles the movement of each vehicle a section at a time. Below are some examples of ModSAF unit and vehicle level tasks:

Unit-level tasks

UActContact Actions on Contact
UAssault Assault
UATAInt Air-to-air Intercept
UBingoFuel Bingo Fuel
UFlwRte Follow Route
UHalt Halt
UOccPos Occupy Position
UPOccPos Pre-occupy Position
UTargeter Targeting
UTraveling Traveling

Vehicle-level tasks

VCollide Backs out of Collisions

VEnemy Assesses the Enemy

VFlwRte Follow Route

VMove Move

VOrbit Orbit

VSpotter Spotter (detects vehicles)

VTargeter Targeting

3.3.1.10 Example Task

The following is an example of a task that monitors collisions and reacts to collisions by trying to resolve them:

```
/* The maximum number of times we'll try to save movemap in a row.
 */
#define MAX_ATTEMPTS 6

static int32 movemap_is_stuck();
static uint32 random_delay();
static void choose_resolution();
static int32 collision_resolved();
static int32 delay_over();
static void do_output();

/* Note that the variables:
 *   int32                vehicle_id;
 *   PO_DB_ENTRY         *task_entry;
 *   TaskStateClass      *state_object;
 *   VCOLLIDE_PARAMETERS  *parameters;
 *   VCOLLIDE_STATE       *state;
 *   VCOLLIDE_VARS        *private;
 * are always available.  Also note that changes to the state data
 * are automatically transmitted on the network.
 */

/* Events:      tick - state machine tick
 *              params - NOP (there are no parameters to this task)
 */

vcollide VCOLLIDE_PARAMETERS VCOLLIDE_STATE VCOLLIDE_VARS

CRITERIA: movement
  TASKPRI_OR(TASKPRI_IN_STATE(delay),
```

```
                TASKPRI_IN_STATE(resolve_collision))
    END_CRITERIA

    '
                    tick()
    '
                            params()
    '
                                    collision(with_whom)
                                        int32 with_whom;
    '
    START
    '
                    private->resolution = VCOLLIDE_IGNORE;
                    private->running_id = PO_OBJECT_ID(task_entry);
                    ^waiting;
    '
    waiting
    '               tick

                    if (!movemap_is_stuck(vehicle_id, private))
                      private->movemap_fix_attempted = 0;
                    else if (private->movemap_fix_attempted <
                            MAX_ATTEMPTS)
                    {
                        private->movemap_fix_attempted += 1;
                        private->collide_id = 0;
                        private->delay = random_delay(private);
                        private->resolution = VCOLLIDE_STOP;
                        ^delay; When stuck, stop moving
                    }

    '                       params
                            ;
    '                               collision

                                    private->collide_id = with_whom;
                                    private->delay = random_delay(private);
                                    private->resclution = VCOLLIDE_STOP;
                                    ^delay; When hit, stop moving
    '
    delay
    '               tick

                    if (movement_enabled)
                      do_output(vehicle_id, private);

                    if (delay_over(private))
                    {
                        choose_resolution(vehicle_id, private);
```

```
                    ^resolve_collision; Back up or ignore
                }

'                       params
                        ;
'                               collision

                                /* Delay some more */
                                private->collide_id = with_whom;
                                private->delay = random_delay(private);
'
resolve_collision
'               tick

                if (movement_enabled)
                  do_output(vehicle_id, private);

                if (collision_resolved(vehicle_id, private))
                {
                    private->resolution = VCOLLIDE_IGNORE;
                    ^waiting; Wait for the next collision
                }

'                       params
                        ;
'                               collision

                                private->collide_id = with_whom;
                                private->delay = random_delay(private);
                                private->resolution = VCOLLIDE_STOP;
                                ^delay; Wait again
'
END
'
                bzero(&private->running_id, sizeof(ObjectID));
'
SUSPEND
'
                bzero(&private->running_id, sizeof(ObjectID));
'
```

After specifying some data structures that will be used by the task, the FSM defines the resource that this task needs in the CRITERIA section above, then the frame which it might push, and the tasks which go into that frame (in this case, there are none). Next come the declaration of the events which this task supports (in this case, the tick, params, and collision events). Then the FSM specifies the things to do when the task is started (or resumed, or migrated from another

simulator). Finally, the FSM defines the behavior called for by each event, in each state. This example has START, waiting, delay, resolve collision, END, and SUSPEND states.

For details about the syntax and structure of FSM files, see the LibTask documentation (see section "AAFSM Code Generator" in *LibTask Programmer's Manual*).

The C source code generated from '`.fsm`' files is compiled and linked together with supporting software to make a task model library. This model is then linked into the application program to become a SAF object subclass. Tasks which need to be driven by events other than the built-in functions `tick` and `params` define the public interfaces to these functions in the task library public header file, and put calls to these functions elsewhere in the simulation software.

3.3.2 Unit Organization

3.3.2.1 Overview

The ModSAF system supports the maintenance of relationships between vehicles and groups of vehicles. This functionality enables one to obtain general hierarchy information (Which subunits and vehicles make up a US M1 company?) as well as hierarchy information about a particular vehicle or unit (Who is this T-72 Platoon's leader and how many subordinates does it have?).

Also included in this functionality is a database which specifies formations of units (wedge formation. column formation, etc.).

Finally, graphics support for unit organization includes the display of unit organization hierarchies in a scrolling window on the GUI and a library that provides services useful for drawing military icons.

3.3.2.2 Related Libraries

The ModSAF libraries that implement unit organization functionality are `liborgdisp`, `libunitorg`, `libechelondb`, `libformationdb`, and `libbgrdb`. They are briefly described here. For more detailed information, refer to the User's Manuals for the respective libraries.

- `liborgdisp` provides a unit organization display service. An application can create an arbitrary number of organization displays and specify the contents to be displayed (UnitClass persistent

objects). `liborgdisp` depends on `libunitorg` to supply the information on how units are organized into a unit hierarchy.

- `libunitorg` manages unit organization information within the SAF simulation. It tracks both the task-organized and function-organized superior and subordinate relationships of units, in order to provide this information without the need for frequent persistent object database queries.

  `libunitorg` also tracks changes to units that are made by outside sources (such as the GUI), and updates the simulation accordingly.

- `libechelondb` provides a database of named standard military echelon organizations (also referred to as "Units"), which can be used as templates or parts of templates for unit creation. `libechelondb` uses a database format which is accessed using `libotmatch` (see section "Overview" in *LibOTMatch Programmer's Manual*). A GUI for unit creation can access `libechelondb` to allow the initialization of a unit to expand into the initialization of an entirely instantiated unit hierarchy. Given a unit to be created, `libechelondb` supplies only the information used to create the unit and its subordinates. It does not actually create the unit persistent object or its subordinate persistent objects.

  The types of information stored in the echelon database are as follows:

  The collection of subunits in a unit.

  The subunits can be recursive references to other libechelondb units. For example, a platoon can contain several vehicles, while a company can contain several command vehicles and several platoons.

  The way vehicle designations are generated for each vehicle or unit.

  For example, a company might be designated as "A", the first platoon in that company might be designated as "A1", and the second vehicle in the first platoon might be designated "A12".

  The order of promotion between units.

  This ordering can also be used to identify unique members in a formation of units. This information is stored implicitly in the data file by the ordering of the subunits.

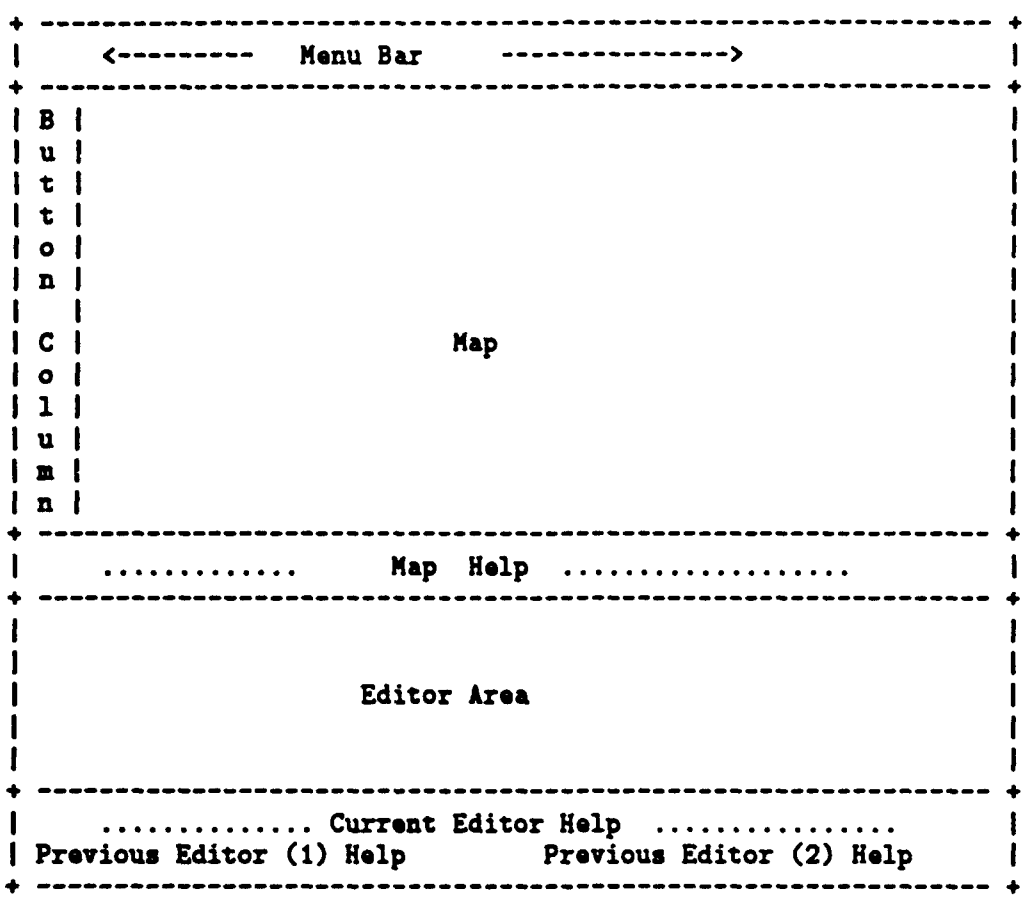
- `libformationdb` provides a database of named standard military formations that can be used for initial placement of units as well as for station keeping of units during movement. `libformationdb` uses a database encoded in `libreader` format to represent the placement of units in a formation. Note that in this context, the term 'units' can refer to individual vehicles or to unit aggregates (such as sections, platoons, companies, etc.).

- `libbgrdb` provides services useful for drawing `libbgr` military icons. `libbgrdb` can perform the pooling and reuse of Pixmaps, as well as the drawing of BGR icons as determined from a database stored in `bgrdb.rdr`.

3.4 The User Interfaces

3.4.1 ModSAF GUI

The Graphical User Interface in ModSAF provides the user with the capabilities to view the current state of the DIS battlefield, as well as create, command and control SAF entities. The following is a layout of the SAFstation screen.



The Map section displays a 2D version of the DIS battlefield. It is also referred as the Plan View Display (PVD). The Editor Area allows users to command and control ModSAF entities and PO objects (such as specifying parameters to an overlay or to a unit, or assigning a unit to a mission). The menubar area at the top of the screen provides functionality for maintaining the ModSAF environment (such as saving a current battlefield scenario or changing the ModSAF user privileges), and alter the appearance of the ModSAF User Interface (such as zooming in or turning on contour lines). The button Column provides two sets of buttons, the object buttons and the map mode buttons. The object buttons allow the user to create ModSAF objects; text, lines, points, vehicles, and to delete objects. The object buttons also provide a tools capability. A vehicle and intervisibility tool is currently implemented with the button tool mode. The map mode buttons

can be used to manipulate the map. There is also two help areas on the screen, the map help provides instructions for mouse clicks in the current map mode, the current editor help provides instructions for the editor fields.

The user interface is a tiled-window system. Each of the different areas of the screen can be resized to take up more or less space. The Editor Area at the bottom of the screen is optionally displayed. The map area can be configured with scrollbars to scroll the map. The GUI provides a mechanism for resizing the editor and map areas.

The user interface architectural support comes from libSAFGUI, which provides the top-level layout, manages the current interface *mode*, and displays help; libSensitive, which allows objects on the map (such as points, or units) to be classified into mouse sensitive groups; libTactMap which provides a 2D view of the terrain database, as well as the ability to add dynamic objects to the map display; libEditor, which provides a means for defining editors in data files.

The following sections describe in more detail the various components of the user interface architecture.

3.4.1.1 Top-level Layout

LibSAFGUI takes care of laying out all the major pieces of the user interface (menubars,message logs, editors, map, etc.) and provides the functions to add and control the user interface. Some of the functionality includes adding to the menu bar, adding a button to the button column, changing the help text, showing or hiding an editor, showing or clearing the message logs, and changing the button mode and button selected.

As explained in the overview, there are two types of buttons in the button column, the object buttons and map mode buttons. Each object button when activated places the ModSAF system in a mode. The new mode may be stacked on top of the current mode, or replace the current mode. This depends on the new mode selected. LibSAFGUI takes care of managing these modes. The types of modes and transitions between modes is described in more detail see Section 7.1 [Overview], page 97.

For the map modes, the system is always in one map mode, and activating a new mode deactivates the previous one.

3.4.1.2 The Terrain Map

There are two major architecture pieces to the ModSAF user interface map. LibTactMap provides a flexible 2D map drawing facility. It draws terrain features as well as other features in the map area. Some of the terrain features that libTactmap can draw are hypsometeric tinting ("altitude by color"), water, roads, trees, buildings, and pipelines. The other features include linear objects, pixmaps, pictures, battefield graphics and intervisibility lines and areas. New map features are added to this library. LibTactmap objects can be made sensitive through the use of LibSensitive, see Section 3.4.1.4 [Sensitive Objects], page 56

LibPVD provides the user will tools to control the Plan View Display. It also takes care of graphically displaying network traffic such as vehicles and weapons fire. The LibPVD controls are built into the libsafgui environment created by the application. The library also takes care of defining the pictures used to draw vehicles of different types. The library uses libTactMap to draw vehicles either as pictures or as libBGR icons (this is controlled through the user interface). The library also provides customization of the interface on a per-terrain-database basis. This is because the best values for things like contour line intervals often differ between databases.

3.4.1.3 The Editor Architecture

Libeditor provides a facility to build flexible data-driven editors with a minimum of effort. The library is an integral part of the ModSAF user interface architecture. An editor consists of many components, called editables. Each one of these editables has a specific representation on the user interface. In a parameter file, the user designates the type of editables contained in an editor, and for each one specifies a field in a data structure with which to store the current value of this editable. The user can also specify which editables, if changed will cause a render function to be called.

An editor is defined by a series of *editables*. Each editable has a name, a type, a storage location (which is a reference back to a name listed in the `struct` part of the data file), and other configuration data. For an example of an editor reader file, see Section 3.1.4.3 [Editor Specifications], page 19 The steps that an application must follow for creating a new editor are:

- The application defines a data structure which is to be edited. This data structure is fixed-size, although it may contain one variable length field (such as the way a protocol PDU cannot be larger than the ethernet packet, but it may contain a variable amount of valid information).
- The application defines a data file which expresses five things about the data which is edited:
  - The name of the editor

  - The memory layout of fields in the data structure.
  - The user interface definition of the editor, providing the label of the editable, the type of editable it is, and the data structure field that this editable will be stored in.
  - The way to initialize each editable
  - The name of storage locations which, when changed, should cause the object to be redrawn, and some other editor specific information.
- The application passes this information to libeditor at startup, at which time a user interface is constructed. Optionally, the application may also pass functions which are called to display the item being edited, and to process the edited item when the editor is exited.
- The application starts the editor when needed.

3.4.1.4 Sensitive Objects

LibSensitive provides a facility for managing mouse-sensitive objects in an X windows environment. A call to libsensitive identifies the sensitive window on the user interface, and identifies X callback routines to be called when various X Events are detected in the window. In ModSAF the map window of the user interface is defined to be sensitive. Objects drawn in the map user interface can be declared to be sensitive. When an object is sensitive, the user can place the mouse over this object and it will be highlighted. Currently in ModSAF, the following objects in the map are sensitive: lines, quadtree objects, bgrs (includes military symbols and vehicles, make this cleared in the libTactMap discussion about the objects in libTactMap and how they are created) , buildings, lines, pictures, pixes, rails, roads, text, trees, water.

3.4.2 The Saf Parser

The SAF Parser is a simple command line interface to ModSAF. It provides a keyboard for testing and debugging of ModSAF operations. With the SAF Parser, you can create unit POs, turn on debugging and off perform remote designation operations, find the launch acceptability for a missile, turn on and off the event monitor. The SAF parser can also perform functions on persistent objects and print out runtime information. You can also modify the simulation rate, read parser commands from a file, and turn on nan overflow, underflow and divide by zero trapping.

3.4.3 The Preview

The preview provides simple "stealth" controls (including free fly, ground hug, and attached modes). The preview display is updated with nearby vehicles, as well as terrain features. It

translates objects to 3D models for a stealth view of the ModSAF battlefield. There is a command line option when running ModSAF to bring one or more previewers. It will come up as a separate window on the workstation.

3.5 Logger

The Data Logger is an application program which records network PDUs. It can record four families of network PDUS; Simnet, DIS, Data Collection and PO. The saved PDUs can be played back. In the case of Simnet and DIS PDUs, the PDUs can be played back in fast-forward, reverse can be frozen, or run at speeds slower or faster than the original simulation speed. When these packets are played back at a different speed, the DIS/Simnet PDUs are modified to perform an approximation of their locations called RVA (Remote Vehicle Approximation).

When the data logger is running, it is constantly listening to the network.

4 Design Methodology

The ModSAF software architecture is an extensible set of software modules which allows rapid development and testing of new agents in the DIS simulated environment. Many ideas for the command and control of automated DIS agents can be implemented and tested without extensive redevelopment of already available SAFOR supporting code. While concepts such as fuzzy logic, reflexive behavior, genetic algorithms, neural networks, expert systems, and cognitive architectures have gained popularity, few have been implemented and tested in large scale real-time simulations. The ModSAF architecture provides a testbed for the evaluation of these intelligent control algorithms within the DIS framework. The SAFOR/Soar project is the first application of this testbed approach.

To be successful, the ModSAF architecture must be able to evolve as the knowledge about building SAFOR systems evolves and as new applications arise. Because there is still much to learn, the architecture must be flexible and expandable while retaining backward compatibility.

Opening the SAFOR system to the DIS community requires a new methodology, which:

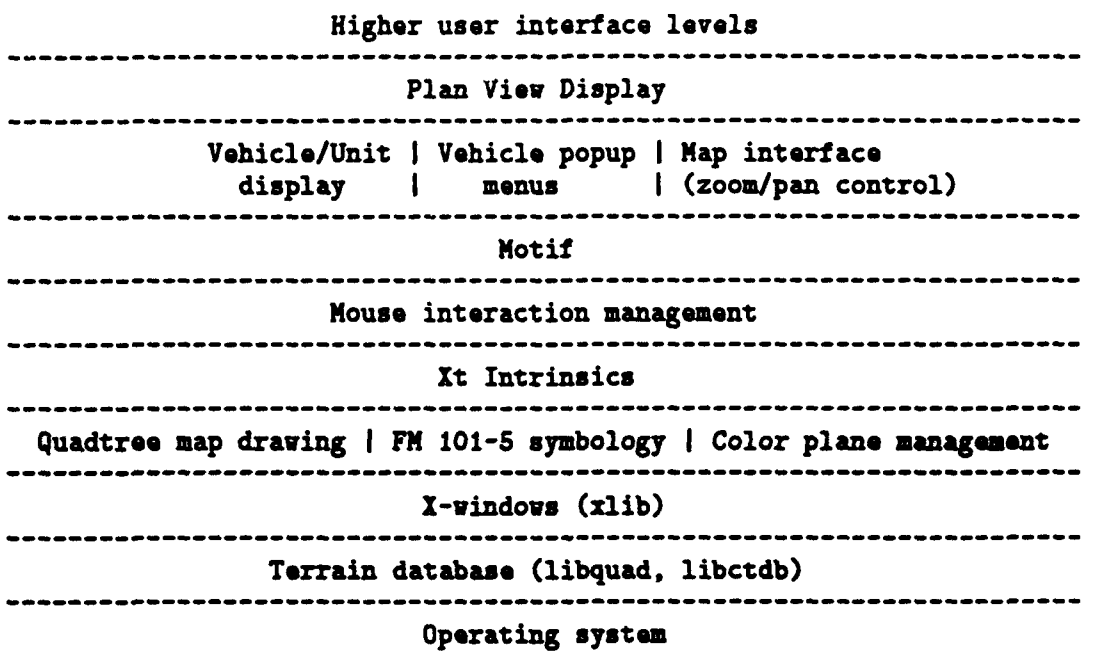
- Allows replacement of individual subsystems without modification of the surrounding software;
- Defines programming practices which will ensure interoperability between independently developed subsystems;
- Allows the use of diverse hardware, which will minimize the buy-in cost to researchers by allowing them to use available hardware;
- Allows subsystems to be written in almost any computer language;
- Allows arbitrary distribution of subsystems across different hardware platforms at run time; and,
- Supports real time, faster than real time, and slower than real time simulation.

4.1 Replacement of Individual Subsystems

To allow easy replacement of individual subsystem modules, the ModSAF architecture takes advantage of four techniques: layering, object based programming, rigorous interface specification, and data driven execution. Although each is completely compatible with the others, they have rarely been used together.

4.1.1 Layering

Layering is the Design Methodoloy used by Dijkstra in the design of the THE operating system, and has been used in most operating systems since. Software modules are grouped into functional layers, and software in one layer is restricted to use only functions and services available in lower layers. This makes software easier to test, since the operation of each successive layer can be confirmed without the having the entire system operational. The use of layering in the ModSAF Architecture allows subsystems to be replaced at varying levels of granularity and facilitates reuse of software. The following figure shows some of the layers involved in the plan view display. Note, the placement of one module above another does not necessarily imply a dependence relationship, rather it only ensures that the lower module does not depend on the module above.



An application can replace the entire Plan View Display subsystem, or just the quadtree map drawing library. The services provided by each software subsystem are well documented, as are the interfaces and dependencies between subsystems.

4.1.2 Object Based Programming

Object based programming is used to cleanly separate the subsystems or modules into classes of objects which are defined by a data structure and a family of functions which operate on that data structure. The data structures of the objects are separated into public and private components

to minimize the interdependence between modules, and to make it easy to change a module's implementation. Single inheritance is supported by building new objects out of subobjects. There is a one to one correspondence between object classes and libraries in the ModSAF system. The use of object based techniques in the implementation of the modules is fully compatible with Ada techniques and ensures compatibility with Ada applications.

Object oriented programming as represented by CLOS, C++, and Objective C goes beyond object based programming by including support from the programming language. The ModSAF architecture is implemented in Kernighan & Ritchie C to maximize compatibility with a variety of hardware platforms, to provide maximum compatibility with other languages by avoiding run time environment requirements, and to make maximum reuse of the large existing body of SAFOR C code. In addition, the use of C generally provides greater or equal run time efficiency to that of other languages. Run Time efficiency is important for SAFOR applications because of the continuing need to tradeoff between cost, numbers of vehicles, and modeling resolution.

4.1.3 Rigorous Interface Specification

To insure the ability of other researchers to replace modules in the ModSAF architecture, the interfaces between object classes (software libraries) are rigorously defined and well documented. It must be clear to a researcher developing a replacement software module exactly what is expected of that module by other software in the system. This documentation is available both in hard copy and on line format and is accessible in electronic form for updating and extension by other researchers. However, in order to maintain compatibility of the system between researchers some centralized configuration management and distribution of changes will be required.

4.1.4 Data Driven Execution

Most ModSAF software modules are heavily data driven. The ModSAF architecture defines objects in its code very broadly. Detailed specification of many objects is actually done at "object creation time" by reading the parameters of the object from a database of object descriptions in main memory. The parameters specify the characteristics and behavior of the object and allow these to be changed without recompiling the system. This is especially important for projects where different variations of the same object are used in different experiments.

An example of data driven execution is the definition of SAFOR vehicles. Items in the data files specify the parametric dynamics model to use (rotary wing aircraft, fixed wing aircraft, tracked ground vehicle, etc.), and the parameters of that model (maximum turn rate, fuel consumption,

etc.). In addition the network representation of the vehicle, and dead reckoning thresholding parameters are specified. Further, the weapons systems are specified by name, and vehicle dependent weapons parameters are given for each (time to load the weapons, accuracy as a function of range, etc.). Since all this data (and much more) is specified for each vehicle, it is possible to change the performance characteristics of each vehicle without modifying any software (as long as the basic algorithms are capable of the desired behavior). Adding new kinds of vehicles often requires only modification of data files.

Another example of data driven execution is the software which draws top-down views of vehicles on the plan view display. A data file specifies the method for drawing various classes of vehicle types by choosing from a small set of graphic primitives (box, line, circle, etc.). Sizes, locations, and rotation (with the hull, or with the turret) information is given for each attribute, as well as the order in which to draw them.

4.2 Programming Practices Guarantee Interoperability

To ensure programming practices support interoperability and module replacement, strict requirements on documentation and referencing of private data are maintained by review and and approval of code entered into the system. The following techniques are used in the ModSAF architecture work to guarantee the protection of private data:

- It is recognized that any public data structure may at least be examined by software in higher levels, and hence the meaning of values should be well documented.
- The modification of public data structures is strictly prohibited, except by way of a function invocation.
- All public header files (where data types and global variables are specified) defined by a library are copied to a public area at compile time, while private header files are never copied from the source area.
- Private functions are defined as 'static' wherever possible (this strictly prevents the compiler from allowing other modules to call them).

4.3 Hardware Independence

The use of K&R C provides a strong starting point for ensuring hardware independence (a K&R C compiler is almost always delivered with any Unix platform). The use of X windows and Motif provides machine independence to the workstation software. Most workstations support

Motif. Operating system independence cannot be guaranteed, since certain operations have never been standardized (getting the value of the real time clock, sending and receiving raw ethernet packets, and other critical features). However, these differences are recognized through the use of conditional compilation (#if sun ... #elseif masscomp ... #elseif mips ... etc.). In practice, the ModSAF system does not use very many operating system calls and therefore there are few conditional sections in the code. Another benefit of using K&R C is that we do not have to pay for a compiler license, reducing the price of development workstations for ModSAF users. As ANSI C matures ModSAF will transition to it.

4.4 Programming Language Independence

It is a primary goal to allow researchers using any language to build upon the ModSAF software. Using C libraries is very flexible because they can be called from virtually any other programming language, but the reverse is often not the case. Programming languages fall into two groups: (1) those that have a run time environment (e.g., Ada, LISP, C++); (2) those which do not (e.g., C, Fortran, Assembly). In general, run time environment languages can't be called by a main program which has no run time environment or a dissimilar run time environment. (Recent changes in Ada compilers are changing this however). If a researcher wanted to install libraries in the system which used a run time programming environment he could do so by translating the main initializing program into the same language. Because almost all the functionality of the SAFOR system has been placed in the software libraries, the small main program body can be translated to any language very easily.

4.5 Distribution of Subsystems Across Hardware Platforms

The base line ModSAF system assumes that the user interface device (sometimes called the *SAFstation*) and the simulation device will probably be run on separate hardware platforms. This is because the real-time constraints of the simulation software may often be violated by the stalling behavior of the X windows system. However, the software can be link-loaded into one executable for systems where hardware is scarce, and accuracy of simulation is not critical.

The management of distributed state information has long been a problem in the SAFOR system. The sharing of vehicle appearance data is handled via the DIS or SIMNET protocols. However, there is a great deal of command and control data which cannot be shared using DIS or SIMNET. The sharing of this data but has been resolved through the invention of the Persistent Object Protocol. This protocol defines classes of objects which can be shared between hardware platforms. Procedures are defined to allow simultaneous editing of objects, and to ensure persistence of objects

despite hardware failures. The protocol is implemented in the ModSAF Architecture through a software library called 'libpo'. Libpo provides an object based database view of the persistent objects, and provides a helpful paradigm for sharing information. The key to the hardware independence of libpo is that when a change is made to the database, callback functions fire on all platforms, including the one which made the change. Hence, software can be written which does not distinguish between "local" and "remote" changes to the database. This allows the simulation of an object to be distributed across several platforms sharing state via the persistent object database.

4.6 Real Time, Faster or Slower Simulation

The ModSAF Architecture relies on a central real time scheduler which invokes function either periodically (for "tick" based simulation), or once, after a specified delay. In addition to this scheduling, functions can access a machine-independent millisecond resolution clock. In order to achieve faster- or slower-than-real-time simulation, the rate of this clock is externally controllable. The persistent object database library also has a real time clock available, which can be used by applications on different hardware platforms as a common, synchronized time reference. Since the persistent object library depends upon the application environment for clock signals, it can be linked to the variable clock just described to achieve a distributed faster-, slower-, or at-real-time speed simulation. (Note that non-SAFOR systems, such as vehicle simulators, will not generally have non-real-time modes, so the use of time altered simulation is somewhat limited.)

5 Extending ModSAF

This section describes how to add new software to ModSAF

5.1 Adding Vehicles

5.1.1 Overview

The ModSAF architecture supports the addition of new vehicles and new vehicle classes. This section describes the procedures for adding new vehicles and vehicle classes, including the files that need to be modified. Throughout this procedure, referring to other libraries and data files will provide examples and helpful hints.

Note: Familiarity with the ModSAF directory structure is assumed. See the Release Notes for more information. In addition, familiarity with makefiles and placing source code under RCS is also assumed.

5.1.2 Adding Vehicles

The following sections describe the necessary modifications to the files responsible for adding a vehicle. The procedure can be broken down into the following steps.

1. Create or modify a vehicle parameter file
2. Make the association between the vehicle and model parameters
3. Define the vehicle
4. Add the vehicle to the master list of vehicles
5. Add the vehicle to the user interface
6. Provide a mapping of vehicle and munitions
7. Add icons for the vehicle
8. Verify the vehicle is in the simnet and DIS protocols

Each step will be explained in more detail in the following sections.

5.1.2.1 Create Parameter File

To define the model parameters, a parameter file needs to be created or better yet is to copy and then modify an existing parameter file. Find an existing vehicle which has characteristics most closely resembling the vehicle you are creating. Then, simply edit the file of the similar vehicle that you copied. For example, to create a M106A1 you can copy the parameter file of the M2 '**src/ModSAF/entities/US_M2_params.rdr**' which is similar and then modify the model parameters accordingly.

```
US_M2_MODEL_PARAMETERS {

(SM_Entity (length_threshold 10.0)
           (width_threshold 10.0)
           (height_threshold 10.0)
           (rotation_threshold 3.0)
           (turret_threshold 3.0)
           (gun_threshold 3.0)
           (vehicle_class vehicleClassTank)
           (guises vehicle_US_M2 vehicle_USSR_BMP)
           (send_dis_deactivate true)
           )
}
```

After you copy this file US_M2_params.rdr to US_M106A1_params.rdr the initial modifications you need to make are to replace all instances of US_M2 with US_M106A1. This file will define all the model parameters for a vehicle that will be used to describe the physical characteristics of your new vehicle. Your vehicle will have the same characteristics as the vehicle from which it was copied until the parameters are modified for your vehicle.

```
US_M106A1_MODEL_PARAMETERS {

(SM_Entity (length_threshold 10.0)
           (width_threshold 10.0)
           (height_threshold 10.0)
           (rotation_threshold 3.0)
           (turret_threshold 3.0)
           (gun_threshold 3.0)
           (vehicle_class vehicleClassTank)
           (guises vehicle_US_M106A1 vehicle_USSR_BMP)
           )
}
```

5.1.2.2 Associate Parameters

In 'src/ModSAF/entities/models.rdr' make the association between the vehicle and the model parameters for that vehicle. This file contains a list of all vehicles, lifeforms, and munitions and they are mapped to their VEHICLE_MODEL_PARAMETERS.

```
("vehicle_US_M106A1"        US_M106A1_MODEL_PARAMETERS GROUND_STD_PARAMS)
```

5.1.2.3 Define the Vehicle

An entry must be created for the new vehicle in '/src/protocol/veh_type.drn'. This file is compiled via the drn compiler to produce 'include/protocol/veh_type.h'.

```
-- M106A1 mortar carrier:
constant vehicle_US_M106A1
    (objectDomainVehicle | vehicleEnvironmentGround |
     vehicleClassSPArmoredTracked | vehicleCountryUS |
     (3 << vehicleSeriesShift) | (3 << vehicleModelShift) |
     vehicleFunctionMortar)
```

This file is compiled using the drn compiler to produce 'include/protocol/veh_type.h'.

```
    /* M106A1 mortar carrier:  */
#define SP_vehicle_US_M106A1            \
        ( SP_objectDomainVehicle | SP_vehicleEnvironmentGround \
         | SP_vehicleClassSPArmoredTracked | SP_vehicleCountryUS \
         | 3 << SP_vehicleSeriesShift | 3 << SP_vehicleModelShift \
         | SP_vehicleFunctionMortar )


#define vehicle_US_M106A1                       SP_vehicle_US_M106A1
```

5.1.2.4 Add to Master List

In 'src/ModSAF/modellist.rdr', an entry for the new vehicle must be added to the master list of all vehicles. The order in which you add is important. You should find a section in the file labelled "add more vehicles here". Add your vehicle entry at the end of that list.

At execution time, ModSAF (main.c) reads in modellist.rdr which contains a list of model files. Modellist.rdr includes generic macro files and specific vehicle parameter files and also models.rdr which contains the association between the vehicles and model data.

```
;; Add more vehicles here
 "US_M109A1_params.rdr"
 "US_M106A1_params.rdr"
```

5.1.2.5 Add to User Interface

In '**/libsrc/libunits/units.rdr**', you must add the vehicle to the unit editor. This maps the vehicle selected on the user interface to the vehicle defined in '**include/protocol/veh_type.h**'.

Note: You will need to remake the affected libraries and link again to see all changes take affect.

5.1.2.6 Map Vehicle to Munitions

In '**libsrc/librdrconst/constants.rdr**', you should verify that a munitions entry and a vehicle entry exists for your vehicle. This provides a gateway between C constant definitions and libreader files.

```
(vehicle_US_M106A1 { 0x28821865 } vehicle_US_M106A1)
```

5.1.2.7 Add Icons

You can specify the icon which will be used to represent your vehicle on the plan view display in '**libsrc/libpvd/pvd.rdr**'. This file defines pictures and maps them to vehicles. They are scaled to size based on the values in physdb.rdr.

```
;; Representative Mortars
(vehicle_US_M106A1 PVD_MORTAR_PICTURES)
```

The '**libsrc/libphysdb/physdb.rdr**' defines the physical attributes of the entity.

```
(vehicle_US_M106A1
 (2.69 4.87 2.64 0.0 22191.0
```

```
3.00 4.87 2.64 0.0 0.0 0.0
0.25 0.25 0.25 0.25 0.25 0.25
0.25 0.25 0.25 0.25 0.25 0.25
0.40 0.40 0.40 0.40 0.40 0.40
0.40 0.40 0.40 0.40 0.40 0.40
 ((primary-turret 0 true 0.15228 -0.53299 1.5 360.0 360.0
    ((main-gun 1 true 0.0 0.0 0.0 0.0 2.0 0.0 0.0 360.0 360.0 0))))))
```

In '/libsrc/libpreview/3Dmodels.rdr', you can specify how your vehicle will look in 3D graphical form.

```
(vehicle_US_M1     ((-1.825 -4.883 0.0) ;; 0
                    (-1.825 4.883 0.0)  ;; 1
                    (1.825 4.883 0.0)   ;; 2
                    (1.825 -4.883 0.0)   ;; 3
                    (-1.825 -4.883 2.375)  ;; 4
                    (-1.825 4.883 2.375)  ;; 5
                    (1.825 4.883 2.375)   ;; 6
                    (1.825 -4.883 2.375)   ;; 7
                    )
                   ((armor 4 5 1 0) ;; Left
                    (armor 5 6 2 1) ;; Front
                    (armor 6 7 3 2) ;; Right
                    (armor 7 4 0 3) ;; Rear
                    (armor 7 6 5 4) ;; Top
                    )
                   )
```

You can specify an entry in 'libsrc/libbgrdb/bgrdb.rdr' which relates object types to military icons which will be displayed for that object (platform). You can also specify whether the icon rotates as the platform rotates.

```
;; Representative Mortars
(vehicle_US_M106A1  PVD_US_MORTAR_ICON)
```

5.1.2.8 Check Protocols

You can verify that an entry exists for your vehicle in '/include/protocol/simnet.mac'. This file provides the mapping between the entity and simnet macros.

```
vehicle_US_M106A1 { 0x28821865 }
```

In '**include/protocol/dis203_entities.h** ', you need to verify that an entry exists for your vehicle.

```
#define SP_DIS203_ ENTITY_SPECIFIC_M106A1                5

#define SP_DIS203_ ENTITY_SUBCATEGORY_ M106A1_MORTAR_CARRIER                9

#define DIS203_ENTITY_SPECIFIC_M106A1  SP_DIS203_ ENTITY_SPECIFIC_M106A1

#define DIS203_ ENTITY_ SUBCATEGORY_ M106A1_ MORTAR_CARRIER
   SP_DIS203_ ENTITY_SUBCATEGORY_ M106A1_MORTAR_CARRIER
```

5.1.3 Adding a Vehicle Class

The following sections describe the necessary modifications to the files responsible for adding a new vehicle class. The procedure can be broken down into the following steps.

1. Create or modify a vehicle class library
2. Add calls to main.c
3. Add calls to libsafobj
4. Modify the Makefile
5. Add the new vehicle class library to modsaf.libs

Each step will be explained in more detail in the following sections.

5.1.3.1 Create Class Library

To add a new vehicle class, a library needs to be created or better yet is to copy and then modify an existing library. Find an existing vehicle class libary which has characteristics most closely resembling the vehicle class you are creating. Then, begin to edit the library of the similar vehicle class library that you copied.

For example, you can copy the library of '**/libsrc/libtracked**' which is similar to the new desired vehicle class and then modify it to get the desired functionality. In addition to changing all the file names to reflect the new library, you also need to change the function names as well. The major change to be made to the new hull library is to add the new dynamics. For libdi, the new dynamics were added to '**/libsrc/libdi/di_tick.c**'.

The following is a list of files contained in the library libdi. Other hull libraries would consist of similar files.

```
Components
Makefile
di_class.c
di_init.c
di_params.c
di_tick.c
libdi.h
libdi.texinfo
libdi_local.h
make.apprules
make.config
make.depend
make.depth
make.docrules
make.driver
make.include
make.librules
```

5.1.3.2 Add Calls to Main.c

Include the new header file of the new library in main.c.

```
#include <libdi.h>
```

Include the call to initialize the new library in main.c.

```
/* dismounted infantry hull */
disinf_init();
```

In addition, the new library name needs to be placed in the 'src/ModSAF/Components' in alphabetical order.

```
libsrc/libdi
```

5.1.3.3 Add Calls to Libsafobj

The '/libsrc/libsafobj/so_local.c ' needs to be modified to include calls to the new library. There are two places in this file to modify.

Include the header file of the new library.

```
#include <libdi.h>
```

Include the call for the new library dynamics routine.

```
    /* Now dynamics */
    disinf_tick(vehicle_id, safobj_ctdb);
```

The '/libsrc/libsafobj/so_init.c ' needs to be modified to include calls to the new library. There are four places in this file to modify.

Include the new header file of the new library.

```
#include <libdi.h>
```

Include the call for the new library collision routine.

```
static void collision_notify(vehicle_id, position, coll_type,
                             other_id, other_mass, other_velocity)
    int32 vehicle_id;
    float64 position[3];
    uint32 coll_type;
    int32 other_id;
    float64 other_mass;
    float64 other_velocity[3];
{
    disinf_collision(vehicle_id, position, coll_type,
                     other_id, other_mass, other_velocity);
    tracked_collision(vehicle_id, position, coll_type,
                      other_id, other_mass, other_velocity);
    wheeled_collision(vehicle_id, position, coll_type,
                      other_id, other_mass, other_velocity);
    missile_collision(vehicle_id, position, coll_type,
                      other_id, other_mass, other_velocity);
    fwa_collision(vehicle_id, position, coll_type,
                  other_id, other_mass, other_velocity);
```

```
        rwa_collision(vehicle_id, position, coll_type,
                      other_id, other_mass, other_velocity);
        vcollide_collision(vehicle_id, other_id);
    }
```

Include the call for the new library damage call.

```
    void safobj_mobility_kill(vehicle_id)
        int32 vehicle_id;
    {
        disinf_damage(vehicle_id, TRUE);
        tracked_damage(vehicle_id, TRUE);
        wheeled_damage(vehicle_id, TRUE);
        fwa_damage(vehicle_id, TRUE);
        rwa_damage(vehicle_id, TRUE);
        ent_set_appearance_bits(vehicle_id, vehMobilityDisabled);
    }
```

Include the call for the new library class init call.

```
        /* First, make the class */
        safobj_class = class_declare_class();

        /* Now, initialize all the class_parts */
        pbt_class_init(safobj_class);
        ent_class_init(safobj_class);
        dsg_class_init(safobj_class);
        disinf_class_init(safobj_class);
        tracked_class_init(safobj_class);
        wheeled_class_init(safobj_class);
```

In addition, the new library name needs to be placed in the '`/libsrc/libsafobj/Components`' in alphabetical order.

```
    libsrc/libdi
```

5.1.3.4 Modify the Makefile

The makefile in '`/src/ModSAF/Makefile`' needs to be modified to include the new library.

```
    LIBS    = \
              -ldi \
```

5.1.3.5 Add to Modsaf.libs

The master list of ModSAF libraries in '`/src/ModSAF/modsaf.libs`' needs to be modified to include the new library.

```
libdfdam
libdi
libdisconst
```

\input texinfo

5.2 Adding Weapons to ModSAF

5.2.1 Overview

ModSAF manages weapons of two categories: guns and missiles. The parser and the tasks are provided a uniform interface in the form of LibGuns, which in turn calls LibBalGun and LibMLauncher. The difference is more apparent when configuring for new weapons. Separate procedures are described here for guns and missiles.

5.2.2 Adding Guns

Guns are modelled by LibBalGun. It must be provided with the following information when adding a new gun to ModSAF:

1. physical characteristics of the gun
2. capacity and performance data to each vehicle using the gun
3. SIMNET-compatible data regarding the ammunition
4. DIS-compatible data regarding the ammunition

The above-listed data falls into the files listed below:

5.2.2.1 physdb.rdr & guns

LibPhysDB manages physical information about vehicles, including their guns. For each gun on the vehicle, there is information about the pivot point and tip point (typically the muzzle), relative to the vehicle's center of mass, as well as the limits on elevation and azimuth adjustment.

The following excerpt is for the US M2 infantry fighting vehicle:

```
(vehicle_US_M2
 (3.20 6.45 2.6 0.0 22590.0
  3.20 6.45 2.6 0.0 0.0 0.0
   ;; Turret information lifted from M1.
   ((primary-turret 0 true 0.15228 -0.53299 1.5 360.0 360.0
      ;; Main gun elevation range is assumed to match that of machine
      ;; gun.  2.032 m barrel??
     ((main-gun 1 true 0.0 0.0 0.0 0.0 2.0 0.0 0.0 1.0472 0.1746 0)
     ;; MG elevation range is +60 to -10 degrees.  545 mm barrel len.
      (machine-gun 2 true 0.0 0.0 0.0 0.0 0.5 0.0 0.0 1.0472 0.1746 0)
     ;; Launcher is a two-tube unit on the left side of the turret that
     ;; deploys vertically.  It has an elevation range of +30 to -20
     ;; degrees.
      (tow-launcher 3 false 0.0 0.0 0.0 0.0 2.0 0.0 0.0 0.5236 0.3491
                    vehTOWLauncherUp))))))
```

Breaking out the gun entry:

```
((main-gun 1 true 0.0 0.0 0.0 0.0 2.0 0.0 0.0 1.0472 0.1746 0)
  |        | |    .---------' .---------' |   |      |      |
  |        | |    |           |           |   |      |      appearance
  |        | |    |           |           |   |      down_limit
  |        | |    |           |           |   up_limit
  |        | |    |           |           length (filled in by code)
  |        | |    |           |tip X,Y,Z
  |        | |    pivot X,Y,Z
  |        | net_represented
  |        artic index
  name
```

In this example, the main gun is a part of the turret, and assigned part number 1. The gun's pivot point is at the vehicle's center of mass. i Its barrel is 2 meters long, so the tip is described as being 2 meters away along the y-axis of the *pivot point's* coordinate system. The barrel can elevate 60 degrees (1.0472 radians) and depress 10 degress (0.1746 radians).

Further explanation of the entire physdb entry is available in (see section "Overview" in *LibPhysDB Programmer's Manual*).

5.2.2.2 vehicle rdr file & guns

The vehicle's reader file contains a LibBalGun configuration for the gun. This includes information about magazine size, the time required to load a magazine, as well as per-munition information such as muzzle velocity, projectile mass and damage tables. Details regarding the LibBalGun entry is available in (see section "Overview" in *LibBalGun Programmer's Manual*).

The following excerpt is from '`US_M2_params.rdr`', which is maintained and installed in the directory '`.../common/src/ModSAF/entities`'. This is the reader file for the US M2 infantry fighting vehicle:

```
  ;; Main gun - M242 25 mm chain gun
([SM_BallisticGun | 0]  (physdb_name "main-gun")
(sensor_name "gunner-sight")
(hit_obscuring_vehicles true)
(rates 0.0 10.0)
(magazine_size 300)
(loading_block 300)
(load_time   8000)
;; M792 is HE/I/T.  Velocity is a guess, based on
;; figure for Bushmaster II (30 mm version).  Stated
;; cyclic rates are 100/200/400 rpm.
(munitions (munition_US_M792
    (round_velocity 1100.0)
    (rate 200)
    (mass 0.185)
    (min_range 0.0)
    (max_range 2200.0)
                HIT_TABLE_MAIN_GUN
                TRACKTIME_TABLE_GENERIC)
  )
)
```

The fields are explained in the documention for LibBalGun. The physdb_name ties the gun to its physical description in '`physdb.rdr`', as explained in the previous section. Note that HIT_TABLE_MAIN_GUN and TRACK_TIME_TABLE_GENERIC are macros.

The vehicle must also be configured with the appropriate number of rounds. This is done through the LibSupplies entry. Again, from the file '`US_M2_params.rdr`':

```
(SM_Supplies (munition_Fuel 662.5);; 175 gallons, in liters.
     (munition_US_M792 600.0);; 25mm HE
     (munition_US_M59 1540.0) ;; 7.62 mm ball
     (munition_US_TOW 5.0);; Hughes TOW
     )
```

In this case, the M2 is configured with 600 rounds of M792 for its main gun. This is the amount that will be supplied by default when an M2 is created.

Finally, the guns must be tied in to the rest of the vehicle via LibComponents. Again referring to the M2:

```
(SM_Components (hull SM_TrackedHull SAFCapabilityMobility)
       (primary-turret [SM_GenericTurret | 0])
       (commander-sight [SM_Visual | 0])
       (driver-sight [SM_Visual | 1])
       (gunner-sight [SM_Visual | 2])
       (main-gun [SM_BallisticGun | 0] SAFCapabilityFirepower)
       (machine-gun [SM_BallisticGun | 1] SAFCapabilityFirepower)
       (tow-launcher [SM_MissileLauncher | 0] SAFCapabilityFirepower)
       )
```

Each gun is included with the bits identifying it as a ballistic gun, and masking in its number within the vehicle (which corresponds to the number used in the LibBalGun entry, above.)

5.2.2.3 mun_type.h & guns

The munitions database is contained in the file 'mun_type.h', which is installed in the directory '.../common/include/protocol'. It is compiled from 'mun_type.drn', which is maintained in the directory '.../common/src/protocol'. For each round, there is is a collection of bits identifying country and caliber. There are also bits that uniquely identify the model and series of the round. There are other bits that characterize the behavior of the projectile upon impact.

The following entry is for the M792 shell, which is used in the main gun of the US M2 infantry fighting vehicle:

```
     /* M792 25mm HEI projectile:  */
#define SP_munition_US_M792        \
        ( SP_objectDomainMunition | SP_munitionClassProjectile \
         | 3 << SP_ammunitionCaliberShift | SP_ammunitionSubclassHE \
         | SP_ammunitionCountryUS | 1 << SP_ammunitionSeriesShift \
```

```
| 0 << SP_ammunitionModelShift )
```

The "SP_" symbols characterize this shell as a high-explosive projectile, less than 30 mm, from the United States.

The model and series are used to further distinguish this from other such HE rounds, and are assigned in the order that rounds of such type are added to the file.

The "SP_" symbols used here are defined in the file '**obj_type.h**', which is installed in the directory '**.../common/include/protocol**'. It is compiled from the file '**obj_type.drn**', which is maintained in the directory '**.../common/src/protocol**'. The only change that might be made to *this* file in defining a new round is the addition of a new subclass.

5.2.2.4 disconst.rdr & guns

The protocol conversion information is contained primarily in the file '**disconst.rdr**'. This file is installed in the directory '**.../common/data**', but is maintained as part of LibDISConst, in the directory '**.../common/libsrc/libdisconst**'. This file maps the SIMNET information for the round from the file '**mun_type.h**' (documented in the previous section) to its DIS equivalent.

The following excerpt is for the US M792 shell, which is used in the main gun of the US M2 infantry fighting vehicle:

```
(munition_US_M792 (DISMunition 3 "UnitedStates" 2 2 0 0
   DISHighExplosiveIncendiary))
```

This entry characterizes the M792 as a high-explosive incendiary round. Note that since the DIS protocol is a carefully-controlled standard, additions must conform to existing definitions. In particular, many of the munitions are already characterized by the standard, and need simply to be added to this file using the defined bit assignments. Refer to the DIS protocol documents for further details.

5.2.3 Adding Missiles

Missiles are covered by two libraries. Missile *launching* is handled by LibMLauncher, while post-launch *modeling* is done by LibMissile. Both present configuration issues when adding a new missile weapon.

To add a new missile to ModSAF, the following information must be added:

1. physical characteristics of the launcher
2. per-vehicle capacity & performance data for the launcher
3. SIMNET-compatible data regarding the missile
4. DIS-compatible data regarding the missile
5. configuration data for the missile itself

The above-listed data falls into the files listed below:

5.2.3.1 physdb.rdr & missiles

LibMLauncher requires information from the physical database regarding the launcher. This information is part of the entry for the vehicle carrying the launcher. This information includes the mounting point relative to the vehicle's center of mass, as well as orientability.

The following excerpt is for the US M2 infantry fighting vehicle:

```
(vehicle_US_M2
 (3.20 6.45 2.6 0.0 22590.0
  3.20 6.45 2.6 0.0 0.0 0.0
   ;; Turret information lifted from M1.
   ((primary-turret 0 true 0.15228 -0.53299 1.5 360.0 360.0
       ;; Main gun elevation range is assumed to match that of machine
       ;; gun.  2.032 m barrel??
      ((main-gun 1 true 0.0 0.0 0.0 0.0 2.0 0.0 0.0 1.0472 0.1746 0)
      ;; MG elevation range is +60 to -10 degrees.  545 mm barrel len.
       (machine-gun 2 true 0.0 0.0 0.0 0.0 0.5 0.0 0.0 1.0472 0.1746 0)
      ;; Launcher is a two-tube unit on the left side of the turret that
      ;; deploys vertically.  It has an elevation range of +30 to -20
      ;; degrees.
       (tow-launcher 3 false 0.0 0.0 0.0 0.0 0.5 0.0 0.0 0.5236 0.3491
                     vehTOWLauncherUp))))))
```

Breaking out the missile launcher entry:

```
(tow-launcher 3 false 0.0 0.0 0.0 0.0 0.5 0.0 0.0 0.5236 0.3491
 |            | |    .---------' .---------' |   |      |
 |            | |    |           |           |   |      |
 |            | |    |           |           |   |      down_limit
 |            | |    |           |           |   up_limit
```

```
|            | |    |           |          length (filled in by code)
|            | |    |           |tip X,Y,Z
|            | |    pivot X,Y,Z
|            | net_represented
|            artic index
name
```

In this example, the TOW launcher is a part of the turret, and assigned part number 3. The launcher's pivot point is at the center of mass. Its housing 1 meter long, but the pivot point is in the middle, so we describe the tip as being 0.5 meters out along the y-axis of the *pivot point's* coordinate system. The launcher can elevate 30 degrees (0.5236 radians) and depress 20 degress (0.3491 radians).

An explanation of the entire physdb entry is available in (see section "Overview" in *LibPhysDB Programmer's Manual*).

5.2.3.2 vehicle rdr file & missiles

LibMLauncher requires information from the vehicle's configuration file, including the name of the launcher, as listed in the file '**physdb.rdr**', magazine capacity, orientability, and types of missiles that may be launched.

The following excerpt is from '**US_M2_params.rdr**', which is maintained in the directory '**.../common/src/ModSAF/entities**'. This is the reader file for the US M2 infantry fighting vehicle:

```
([SM_MissileLauncher | 0]  (physdb_name "tow-launcher")
   (sensor_name "")
   (rates 0.0 0.0)
   (magazine_size 2)
   (load_time   27000)
   (launcher_time 5000)
   (movable true)
   (track_time 5000)
   (munition 2 munition_US_TOW))
```

The fields are explained in the documention for LibMLauncher. The **physdb_name** field ties the launcher to its physical description in '**physdb.rdr**', as explained in the previous section.

The vehicle must also be configured with the appropriate number of missiles. This is done through the LibSupplies entry. Again, from the file 'US_M2_params.rdr':

```
(SM_Supplies (munition_Fuel 662.5);; 175 gallons, in liters.
     (munition_US_M792 600.0);; 25mm HE
     (munition_US_M59 1540.0) ;; 7.62 mm ball
     (munition_US_TOW 5.0);; Hughes TOW
     )
```

In this case, the M2 is configured with 5 TOW missiles. This is the number of missiles that will be supplied by default to an M2 when it is created. Details regarding LibSupplies are available in (see section "Overview" in *LibMLauncher Programmer's Manual*).

Finally, the missile launchers must be tied in to the rest of the vehicle via LibComponents. Again referring to the M2:

```
(SM_Components (hull SM_TrackedHull SAFCapabilityMobility)
       (primary-turret [SM_GenericTurret | 0])
       (commander-sight [SM_Visual | 0])
       (driver-sight [SM_Visual | 1])
       (gunner-sight [SM_Visual | 2])
       (main-gun [SM_BallisticGun | 0] SAFCapabilityFirepower)
       (machine-gun [SM_BallisticGun | 1] SAFCapabilityFirepower)
       (tow-launcher [SM_MissileLauncher | 0] SAFCapabilityFirepower)
       )
```

Each missile launcher is included with the bits identifying it as a missile launcher, and masking in its number within the vehicle (which corresponds to the number used in the LibMLauncher entry, above.)

5.2.3.3 mun_type.h & missiles

The munitions database is contained in the file 'mun_type.h', in directory '.../common/include/protocol'. It is compiled from the file 'mun_type.drn', which is maintained in the directory '.../common/src/protocol'. The entry for a missile includes bits identifying the country, intended target type, and warhead type, as well as an assigned model and series (e.g. M1A1 vs. M1A2).

The following entry is for the US TOW missile:

```
#define SP_munition_US_TOW          \
```

```
( SP_objectDomainMunition | SP_munitionClassMissile \
 | SP_missileTargetArmor | SP_missileWarheadShapedCharge \
 | SP_ammunitionCountryUS | 1 << SP_ammunitionSeriesShift \
 | 0 << SP_ammunitionModelShift )
```

The "SP_" symbols characterize the TOW as an anti-tank missile with a shaped-charge warhead, of American manufacture.

The model and series are used to further distinguish this from other such missiles, and are assigned in the order that missiles of such type are added to the file.

The "SP_" symbols used here are defined in the file '**obj_type.h**', which is installed in the directory '**.../common/include/protocol**'. It is compiled from the file '**obj_type.drm**', which is maintained in the directory '**.../common/src/protocol**'. The only change that might be made to *this* file in defining a new round is the addition of a new subclass.

5.2.3.4 disconst.rdr & missiles

The protocol conversion information is contained primarily in the file '**disconst.rdr**'. This file maps the SIMNET information for the munition from the file '**mun_type.h**' to its DIS equivalent. It is installed in the directory '**.../common/data**', but is maintained as part of LibDISConst in '**.../common/libsrc/libdisconst**'.

```
(munition_US_TOW (DISMunition 2 "UnitedStates" 1 1 0 0
   DISHighExplosiveShapedCharge))
```

Details of this data structure are in the documentation for LibDISConst.

This entry characterizes the M792 as a high-explosive incendiary round. Note that since the DIS protocol is a carefully-controlled standard, additions must conform to existing definitions. In particular, many of the munitions are already characterized by the standard, and need simply to be added to this file using the defined bit assignments. Refer to the DIS protocol documents for further details.

5.2.3.5 missile reader file

Once launched, a missile is modelled as a vehicle distinct from the vehicle that launched it. Ac-

cordingly, there is a separate configuration file for the missile vehicle. This file contains information regarding range, speed, and any sensor used in guiding it in flight.

The following is the contents of 'US_TOW_params.rdr', the vehicle configuration file for the TOW missile carried by the US M2. This file is installed and maintained in the directory '.../common/src/ModSAF/entities'.

```
US_TOW_MODEL_PARAMETERS {

(SM_PBTab)
(SM_Entity (length_threshold 10.0)
   (width_threshold 10.0)
   (height_threshold 10.0)
   (rotation_threshold 3.0)
   (turret_threshold 3.0)
   (gun_threshold 3.0)
   (vehicle_class vehicleClassSimple)
   (guises munition_US_TOW munition_US_TOW)
   (send_dis_deactivate false))

(SM_Collision (check buildings platforms ground trees)
      (announce buildings)
      (duration 0)
      (feature_mass 10000.0)
      (fidelity high))

(SM_Components (hull SM_MissileHull SAFCapabilityMobility))

(SM_MissileHull (sensor_name "")
                (sensor_on_board false)
                (parent_sensor_name "")
                (pursuit_mode lead_pursuit)
(simulation munition_US_TOW)
                (range 3750.0)
                (launch_speed 5.0)
                (acceleration 250.0)
                (safe_time 0.25)
                (loal_time 0.1)
                (max_burn_time 15.5)
                (burn_max_turn 5.0)
                (coast_max_turn 5.0)
                (directionality 12.566370614359)
;; BGM-71* go 278 m/s
(max_speed (0.0 0.91)
   (20000.0 0.91))
)
}
```

What primarily distinguishes a missile vehicle file from a true vehicle file is the SM_MissileHull entry, which is used by LibMissile. Details can be found in the LibMissile documentation.

5.3 Adding Tasks

5.3.1 Overview

The ModSAF architecture supports the addition of new tasks. This section describes the procedure for adding new tasks, including the files that need to be modified for the addition of a new task. Throughout this procedure, referring to other task libraries will provide examples and helpful hints. Some of the current tasks are `SM_VSpotter`, `SM_UHalt`, and `SM_UTravel`.

Note: Familiarity with the ModSAF directory structure is assumed. See the Release Notes for more information.

5.3.2 Procedure

The following sections describe the necessary modifications to the files responsible for adding a task. The procedure can be broken down into the following steps.

1. Add a SAFModel for the task to '`p_safmodels.h`'
2. Use '`osatemplate`' to generate a basic library
3. Add extra components to the task library
4. Register the task with libsafobj
5. Modify '`main.c`', '`Makefile`', '`modsafdir.texinfo`', '`modsaf.libs`'
6. Add a taskframe to '`taskframes.rdr`'
7. Make additions if defining a Reactive Task

Each step will be explained in more detail in the following sections.

5.3.2.1 Add SAFModel

To add a SAFModel, the file '**common/include/protocol/p_safmodels.h**' needs to be modified. To do this, simply edit the file and add a task at the end. For example, if the last defined tasks of '**p_safmodels.h**' looks like:

```
#define SP_SM_UReactIF              ( SP_SM_classTask | 49 )
#define SP_SM_UReactNoMission       ( SP_SM_classTask | 50 )
#define SP_SM_VCollide              ( SP_SM_classTask | 51 )
```

then the addition for the new task should look like:

```
#define SP_SM_NewTask               ( SP_SM_classTask | 52 )
```

The string *NewTask* should be replaced with descriptive name for the new task.

5.3.2.2 Generate Basic Library

The '**osatemplate**' tool generates a basic template library. To use this tool, cd to the '**common/libsrc**' directory and execute the command '**osatemplate**'. It will prompt you as follows:

```
osatemplate prompt>    module name (e.g., entity, pbtab)?
Your Respose>             newtask

osatemplate prompt>    file prefix (e.g., ent, pbt)?
Your Respose>             nwtsk

osatemplate prompt>    function prefix (e.g., ent, pbt)?
Your Respose>             nwtask

osatemplate prompt>    SAF model number (e.g., SM_Entity, SM_PBTab)?
Your Response>            SM_NewTask
                              /\
                               ---------- Note:  This needs to be
                                          the same as defined above
                                          in 'p_safmodels.h'
```

In this case, the file prefix was chosen to be *nwtsk* because there is a maximum name length of 15 for '.c' and '.h' files on some platforms. The function prefix was chosen to be *nwtask* to show

that it does not necessary have be the same as the file prefix or module name. Many times the module name and the function prefix use the same string.

Replace the strings *newtask*, *nwtsk*, *nwtask*, and *SM_NewTask* with the appropriate names for the desired task. The '**osatemplate**' tool will generate a library with the previously entered strings in the appropriate places. The files will contain the word *TEMPLATE* throughout the library. With the word *TEMPLATE* is an explanation of what needs to replace it. These *TEMPLATE* occurrences need to be replaced with the appropriate code or comments.

5.3.2.3 Library Components

The required library components for a task are as follows:

```
nwtsk_params.c
nwtsk_class.c
nwtsk_init.c
nwtsk_task.fsm
nwtsk_util.c
newtask.rdr
libnewtask.h
libntsk_local.h
libnewtask.texinfo
Makefile
```

Most of these files are generated by the '**osatemplate**' script. The files that are not are: '**nwtsk_task.fsm**', '**nwtsk_util.c**', and '**newtask.rdr**'.

The file '**nwtsk_task.fsm**' is the task state machine written using the AAFSM format. This file is translated to C using the '**fsm2ch**' utility (see section "Libtask" in *LibTask Programmer's Manual*). For information about how a task state machine will work refer to Task information (see section "Tasks" in *Behavior Models*).

The file '**nwtsk_util.c**' contains one necessary function, '**nwtask_init_task_state**'. This function initializes the size, refcount, and state variables of the task class state object for this task. This po structure represents the current state of the task. (see section "Libpo" in *LibPO Programmer's Guide*).

The other file that is not produced by the '**osatemplate**' script is the '**newtask.rdr**' file. This file is what the libeditor library (see section "Libeditor" in *LibEditor Programmer's Manual*) uses

to build the task editor. Look at another task library's 'rdr' file to get an idea of what needs to be in there.

A couple of other things need to be done that are specific to task libraries. These are:

1. Register the editor file, '**newtask.rdr**', and initialize the task state function with the taskedit library in the **nwtask_init** function. The initialization routine is called when this task gets added to an entities' current frame (see section "Libtaskedit" in *LibTaskEdit Programmer's Manual*).
2. Register a status function with the statmon library in the **nwtsk_init** function. This is done so that useful messages can be displayed in the status monitor during execution of the modsaf process.

5.3.2.4 Registering

Libsafobj handles the initialization, creation, and deletion of the SAF vehicle subclasses. When a new task is added, the function **nwtask_class_init** must be called from '**so_init.c**'. This will initialize the new vehicle task subclass. Also, the **nwtask_create** function and the **nwtask_destroy** function must be added to the file '**so_local.c**'. The create routine is called when a vehicle is created to create this task subclass. Likewise, the destroy routine is called when a vehicle is destroyed to destroy this task subclass. Search for another task, such as **libuhalt**, in these files for an example.

5.3.2.5 Other Modifications

When adding a new task there are certain files in '**common/src/ModSAF**' that need to be modified. These files are:

'**Makfile**' In the '**Makefile**', the new task library needs to be added to the libraries that are linked.

'**main.c**' '**Main.c**' must call the library init function, **nwtask_init**, for the new task.

'**modsaf.libs**'
The new task library needs to be added to the list of modsaf libraries in the file '**modsaf.libs**'. This file is used to build all the libraries in ModSAF.

'**docs/modsafdir.texinfo**'
A reference to the new task documentation needs to be added to the modsaf documentation directory list in the file '**docs/modsafdir.texinfo**'.

'entities/standard_params.rdr'

Each unit has a list of the tasks and parametric data that it can use. This information can be found in the 'rdr' files in 'common/src/ModSAF/entities'. If the task has the same parametric data for every unit, it should be added to the file 'standard_params.rdr'.

5.3.2.6 Add Taskframe

If the task is a vehicle level task, it is usually put into the background frame or it is spawned by a unit level task. To put a vehicle level task in the background taskframe, add the function **nwtask_set_background** into the '**nwtsk_class.c**'. See the library libvmove for an example of how to put a vehicle level task into the background taskframe.

If the task is a unit level task, it needs to be included in a taskframe. To create a new taskframe, '**taskframes.rdr**' should be modified. In the '**taskframes.rdr**' file, a text string specifies the name of the taskframe followed by the tasks contained in the taskframe. The following is an example of the taskframe *Move(Hold)*.

```
("Move (Hold)" SM_UHalt SM_UTravel

 (SM_UHalt PREPARATORY
       (prep_var CONSTANT 1))

 (SM_UTravel ACTUAL
       (route FORCE "You must specify a route (point, text, or line)")
       (speed CONSTANT 8.0)
       (speed_limit CONSTANT 0.0)
       (formation CONSTANT wedge)
       (roadmarch CONSTANT 0)
       (conform CONSTANT 0)
       (form_type CONSTANT 0))

(SM_UActionOnContact BOTH
       (engagement_range CONSTANT 2000.0)          ; 3500 meters
       (under_fire_enemy_threshold CONSTANT 3.0)   ; 3 enemy vehicles
       (under_fire_small_enemy_action CONSTANT
                                      UACTCONTACT_ACTION_NONE)
       (under_fire_large_enemy_action CONSTANT
                                      UACTCONTACT_ACTION_NONE)
       (not_under_fire_enemy_threshold CONSTANT 3.0) ; 3 enemy vehicles
       (not_under_fire_small_enemy_action CONSTANT
                                          UACTCONTACT_ACTION_NONE)
       (not_under_fire_large_enemy_action CONSTANT
                                          UACTCONTACT_ACTION_NONE)
```

```
        (fire_technique CONSTANT 1))                    ; alternating fire

    (SM_UEnemy BOTH)
    )
```

There are four unit level tasks in the above taskframe, SM_UHalt, SM_UTravel, SM_UEnemy, and SM_UActionOnContact. The keyword *PREPARATORY* specifies the task is part of the preparatory frame. The keyword *ACTUAL* specifies the task is part of the actual frame. The keyword *BOTH* specifies the task is part of both the preparatory and actual taskframe. For more information on preparatory and actual taskframes see the section on taskframes (see section "Task Frames" in *Behavior Models*).

The parameters to each task in a taskframe are listed with the task. Each of the tasks is listed with the initialization values for its data structure. This should match the *initial* section in the **newtask.rdr** file in the task library. For example, the following is part of the 'utrav.rdr' file from the SM_UTravel library.

```
(initial (route FORCE)
         (speed CONSTANT 8.0)
         (speed_limit CONSTANT 0.0)
         (formation CONSTANT wedge)
         (roadmarch CONSTANT 0)
         (conform CONSTANT 0)
         (form_type CONSTANT 0)
         )
```

Notice that the data is the same as in the taskframes.rdr file above. The default values can be different. The default values that are entered in the 'newtask.rdr' are overridden by the default values entered in the file 'taskframes.rdr'.

5.3.2.7 Reactive Tasks

Reactive tasks are similar to non-reactive tasks. The few differences are: reactive tasks are never suspended when spawning an opaque taskframe, but instead reactive tasks are able to monitor the state of the reactive task frame pushed and stop it if necessary. Refer to the Task documentation for more information (see section "Reactive Tasks" in *Behavior Models*).

Therefore, while writing the task state machine for a reactive task, the above points need to be considered and implemented appropriately.

For an example of a reactive task, refer to the `libuactcontact` reactive task library.

5.4 Adding Echelons

* Overview:: Overview of Echelon Layers in ModSAF * Files:: Steps to Creating a New Layer of Echelon

5.4.1 Overview

ModSAF supports the addition of layers of echelon for existing entities. For example, assume the entity "M109 Howitzer" exists, as a stand alone vehicle. The capability exists to produce Platoons, Companies, Batallions, etc. of M109's.

New layers of echelon are created by modifying three files, distributing the modified files, and "remaking" ModSAF. The three files which control echelon creation are:

```
../libsrc/libechelondb/echelondb.rdr

../libsrc/libunits/units.rdr

../src/protocol/unit_type.drn
```

Note: Familiarity with the ModSAF directory structure is assumed. See the Release Notes for more information.

5.4.2 Files

The following sections describe the necessary modifications to the files responsible for defining layers of echelon. Each section represents the file to be modified and provides an example and explanation of the type of entry required.

* echelondb.rdr:: Defines the unit and its leaf components * units.rdr:: Defines the X-interface menu to unit mapping * unit_type.drn:: Defines the system constant for the unit

5.4.2.1 echelondb.rdr

```
(unit_US_M106A1_Platoon ((leaf vehicle_US_M106A1 "??1")
                         (leaf vehicle_US_M106A1 "??2")
                         (leaf vehicle_US_M106A1 "??3")
                         (leaf vehicle_US_M106A1 "??4")))
```

echelondb.rdr defines the unit and its leaf components. The term "leaf" means that this is a terminal node in the unit hierarchy. If the subunit can be expanded into other subunits, the term "tree" would be used. The above example shows how to define a Platoon and its components for an M106A1. A more in depth discussion of the physical "leaf" and "tree" definition can be found i n the Echelondb Programmer's Reference Manual.

The file echelondb.rdr, in the ../libsrc/libechelondb directory needs to be mod ified. Add a unit definition for your layer of echelon.

After the file has been modified, it will be necessary to issue a "make headers" from ../libsrc/libechelondb to distribute the reader file.

5.4.2.2 units.rdr

```
("M106A1    Platoon" unit_US_M106A1_Platoon)
```

units.rdr defines the X-interface menu string which will be displayed in the pull-down menu for "Unit Type", when creating a new entity with the units editor. It maps this selection to the symbolic tag "unit_US_M106A1_Platoon", which defines the entity.

The file ../libsrc/libunits/units.rdr needs to be modified. Add an entry containing:

1. **The string you wish to use in the pull-down menu.**
2. **The symbolic unit tag that you defined in echelondb.rdr**

After the file has been modified, it will be necessary to issue a "make headers" from ../libsrc/libunits to distribute the new reader file.

5.4.2.3 unit_type.drn

```
-- M106A1 units

   constant unit_US_M106A1_Platoon
            (objectDomainUnit | unitEnvironmentGround | unitCountryUS |
             unitSizePlatoon | unitRoleArmor)
```

unit_type.drn defines the system constant for the given unit. This constant is used throughout ModSAF to identify this unit type. The above entry creates a constant definition for a US_M106A1_Platoon unit.

The fields in the definition (objectDomainUnit, etc.) are symbolic constants which map to bit settings, which in turn, define particular characteristics of the vehicle. A list of available characteristic settings is defined in ../common/src/protocol/obj_type .drn.

The file ../src/protocol/unit_type.drn needs to be modified. Add a "constant" entry for your layer of echelon.

IMPORTANT: You should verify that no other entries exist with the same definitions as your new entry. If so, you will need to add " | <num>" after the last descriptor. For example if you found that the characteristics existed for the example entry above, you would need to replace "unitRoleArmor" with "unitRoleArmor | 1" in the existing entry, and replace "unitRoleArmor" with "unitRoleArmor | 2" in your new entry.

If " | <num>" exists in a current entry, then increment <num> in your entry.

Note, after you have modified this drn file, you will need to compile it with the drn compiler, by issuing "make unit_type.h" from ../src/protocol.

Then, copy the new unit_type.h file to ../common/include/protocol. Next, from librdrconst, issue "make clean install" to build the new constants file. Now, ModSAF can be run, and the new layer of echelon will be present.

6 Sample M1 Tank Execution

This section describes the tick routines that are called for an M1 tank when it is created and after it is assigned a task to move on contact and assault an enemy vehicles it encounters. The tick routines are broken down into the physical and the behavioral models.

6.1 Physical Models

The following is a sample of the physical model routines that are called when an M1 tank is created. From the parameter file for an M1 tank (US_M1_params.rdr and standard_params.rdr), we see that the M1 tank consists of many vehicle subclasses. Each of the vehicle subclasses has an associated tick routine that is called for each vehicle. The vehicle subclasses for an M1 are:

```
From the US_M1_params.rdr:

entity - provides network entity appearance information
dfdamage - a direct fire damage model
ifdamge - a indirect fire damage model
components -
        a Tracked Hull
        a generic turret
        three visual sights, the driver, the loader and the gunner
        two guns, a main-gun and a machine-gun

From the standard_params.rdr GROUND_STD_PARAMS section:

pbtab - position based table
localmap - provides vehicle management of movement
taskmanager - manages the tasks
collision - provides a 3D physical model of collision detection
genradio - provides rudimentary radio communications
```

When an M1 tank is created, all of the models or vehicle subclasses call their create routines to create a new instance of this model. Then the models start ticking. The following tick routines are added to the scheduler for this new vehicle:

```
ent_tick - entity tick
dfdam_process_pdus - process direct fire damage pdus
ifdam_process_pdus - process indirect fire damage pdus
tracked_tick - tracked hull tick
generic_turret_tick - the generic turret component
```

```
visual_tick - one for eahc of the visual sights
bgun_tick - one for each of the guns

pbtab_tick - position based table tick
localmap_tick - movement management tick
taskmgr_tick - task manager tick
coll_tick - collision tick
grad_tick - generic radio tick
```

6.2 Behavioral Models

When a new vehicle is created, some background tasks start running for this new vehicle. The files '`US_M1_params.rdr`' and '`standard_params.rdr`' list the tasks that will run in the background upon vehicle creation. The reader files have the keywords background on for the specific tasks that are to run in the background. The following are the tasks that are started when an M1 tank is created.

```
vspotter - a vehicle spotter task
vassess - a vehicle assessment task
vsearch - a vehicle seach task
venemy - a vehicle enemy task
vmove - a vehicle move task
vterrain - a vehicle terrain task
vcollide - a vehicle collision task
```

Each of these vehicle subclasses has a tick routine defined which will be added to the scheduler when a new vehicle is created.

When the M1 tank is assigned a move-contact task frame and told to assault any enemy vehicles it comes in contact with, the following additional tasks start ticking. Since the vehicle level tasks are running in the background for this example, the only role the vehicle is playing on the task frame stack is as a unit.

The uactoncontact and uhalt get pushed on as transparent task frames, along with a uhalt. The uhalt is the opaque task frame at the top of the stack.

```
                Role            Task Frame Stack
                =====           ================
prepartory:     Unit            Halt (tasks:uactoncontact, uenemy, uhalt)
```

Since this is not an on-order, the uhalt immediately gets popped off the stack and a utraveling gets popped on. The utraveling just gives its parameters to vmove to do the actual moving.

```
                Role            Task Frame Stack
                =====           ================
actual:         Unit            Move (tasks:
                                              uactoncontact
                                              uenemy
                                              utraveling)
```

While the vehicle is moving, it encounters an enemy. The task frame stack pushes on an assault task frame. One of the tasks in this task frame is the sponsoring task, which in this case is uactoncontact. This task continues to run when the assault task frame is pushed on the stack. The assual task frame uses a pointer back to the uactoncontact in the move task, instead of making a copy of the task. This is to ensure that if any parameters where changed for the sponsoring task they would be identified when it was part of the assault task frame.

```
                Role            Task Frame Stack
                =====           ================
actual:         Unit            Move (tasks:
                          ------------------>   uactoncontact
                         |                      uenemy
                         |                      utraveling)
                         |      Assault (tasks:
                         |                      uassault
                         |                      uenemy
                         |                      utargeter
                         |                      utraveling
                         ---------------------- sponsoring task)
```

When the vehicle has finished assaulting the vehicle, the assault task frame will be popped of the stack and the task frame stack will return back to:

```
                Role            Task Frame Stack
                =====           ================
actual:         Unit            uactoncontact - transparent task frame
                                uenemy - transparent task frame
                                utraveling - opaque [TOP OF STACK]
```

To see more information on the transparent and opaque task frames see Section 3.3.1.3 [Task Frames], page 40

7 Memory and Processing Time

7.1 Overview of Benchmarking

Benchmarking measures software performance in a way that serves as a standard by which other similar software may be measured. For example, benchmarking can measure the time spent during I/O calls for different protocols, and the maximum number of local and remote vehicles that can be simulated with good performance. These results will hopefully indicate where the software performs very well and where some improvements can be made. The results can also be compared with results of similar software to see which product gives better performance.

7.2 Setup environment and suggestions

To perform successful benchmarking tests with modsaf:

- Compile two versions of the source, a SAFsim and a SAFstation. The SAFstation can be compiled normally, and run with the -nosim option. The SAFsim should be compiled with an optomizer.
- The SAFsim should be compiled *WITHOUT* "-DUSE_X -DUSE_MOTIF -DHYPSO". Make sure these flags are not in the environment variable EXTRA_CFLAGS by typing "printenv EXTRA_CFLAGS" at the shell prompt.
- Provided that testing is being done on R4000 or R4400 SGIs, the SAFsim should be compiled *WITH* "-mips2". This can be added to EXTRA_CFLAGS, or to tools/make.config.
- Run the front end and back end (SAFsim and SAFgui) on different machines.
- Do NOT run with the profiler.

Each of these items will speed up the performance of the system. For example, X and Motif code does not need to be compiled for the SAFsim since it will not use any X or Motif. This will make the code more efficient. The profiler will slow the system down somewhat so be sure not to use it when benchmarking.

7.3 How to benchmark

The following sections describe how to do some benchmarking tests with ModSAF.

7.3.1 Network

A test can be performed that measures network performance. The times for reads and writes on the network for the SIMNET protocol can be compared to the times of reads and writes for the DIS protocol. The average time spent writing or reading packets on the network can be calculated by inserting some timer routines, and then calculating the average time spent per call. The timer routines should call gettimeofday(3) to get the time in microseconds. When running with the simnet protocol (-simnet), the procedures pv_assoc_read and pv_assoc_write are used to send and receive packets on the network. These procedures are in libpktvalve/pv_assoc.c When running with the dis protocol (-dis), the procedures pv_udp_read and pv_udp_write are used to send and receive packets on the network. These procedures are in libpktvalve/pv_udp.c

See Results (see Section 7.5.1 [Network results], page 101) for more information.

7.3.2 Vehicle limit

Vehicle limit for local vehicles:

A test can be performed to determine the maximum number of local vehicles that are able to be simulated with good performance. Good performance means that 90% of the ticks in the 67 ring are under 500ms. The message "Gasp! Out of Cycles!" appears when less than 90% of the ticks in the 67 ring are under 500ms (ie. when performance is not good anymore). The message "Sigh. All better." appears when the load has gone back down to an acceptable level.

Loading from a scenario may cause a temporary overload which is okay provided that the load goes back down in a minute or two. If a scenario was loaded then give the safsim a minute or two before checking for the Gasp message.

The test should be performed with local invincible vehicles that have unlimited ammunition and are continuously moving. Unlimited ammunition can be set in the Unit editor by typing either 'U' or 'Unlimited' in the number box next to each munition for the vehicle. To have the vehicles constantly moving and shooting, an octagonal spiral route on flat ground is recommended. The setup should be a safsim running on an unloaded CPU connected via network with very little traffic to the safgui. Vehicles should be added a platoon at a time, or one at a time until the Gasp message appears.

Vehicle limit for remote vehicles:

A ModSAF-based packet blaster is needed to test the amount of remote vehicles a SafSim can handle with good performance. A packet blaster is a program that sends packets out on the network at a specified rate that is equivalent to having x remote vehicles. The default number of remote vehicles is 800, but this can easily be changed at the command line. The blaster simulates vehicles in clumps of four at random locations all over the terrain, and sends each clump in random directions at a random velocity, along the terrain.

The blaster program is located in src/blaster and has options similar to modsaf (-simnet/-dis, -version -udp/-assoc, -exercise, etc.). There is also an option for specifying the number of remote vehicles to be simulated (-vehicles x) and an option for specifying the number of vehicles per clump (-clump x). These options may be specified at the command line or in the environment variable BLASTERARGS.

When the blaster program is running on the same exercise as a safsim, clumps of moving vehicles will appear on the safsim in random places. One way to measure the SafSim's performance is to use the "monitor" command at the command line in the saf parser. Bad performance begins when 90% of the ticks in the 67 ring are under 500ms. The other way to measure the performance is to keep increasing the number of remote vehicles until the message "Gasp! Out of Cycles!" appears. This message appears when 90% of the ticks in the 67 ring are under 500ms.

See Results (see Section 7.5.2 [Vehicle limit results], page 101) for more information.

7.3.3 Protocol family traffic

A test can be performed to measure the percentages of different packet types that are sent out on the network. When a packet is to be sent out on the network, the procedure pv_write_packet in libpktvalve/pv_io.c is called. Code can be inserted in here to keep track of the number of calls and the packet protocol family type. A list of the different types of calls are in include/protocol/p_num.h

Note: The PO packet rate will be very high in the beginning and will become steadier once the sim knows about all of the objects. The simulation packets will increase during the rest of the test.

See Results (see Section 7.5.3 [Protocol traffic results], page 102) for more information.

7.3.4 Future

This is a list of other tests that should be done in the future.

- Test modsaf with more than one sim to see if there is an improvement, and to see if load sharing works correctly.
- Test the packet rate for an idle group of vehicles. Supposedly, if the vehicles have been idle for a while then packets will not get sent out until they are requested from another sim. This should allow for more vehicles.

7.4 Profiling

Profiling is useful to find out which procedures are being called the most and which procedures are taking up the most CPU time. There are two kinds of profiling: pc-sampling and basic-block counting. Pc-sampling interrupts the program periodically, recording the value of the program counter. Basic-block counting divides the program into blocks delimited by labels, jump instructions, and branch instructions. It counts the number of times each block executed. This provides more detailed (line by line) information than pc-sampling.

Using pc-sampling:

- compile modsaf with -p
- run modsaf (generates "mon.out")
- type "prof modsaf mon.out"

Using basic-block counting:

- compile modsaf the regular way (without -p).
- type "pixie -o modsaf.pixie modsaf" (generates "modsaf.Addrs")
- type "modsaf.pixie" (generates "modsaf.Counts")
- type "prof -pixie modsaf modsaf.Addrs modsaf.Counts"

To learn more about the commands and to help interpret the output from the profiler, refer to the man pages. Prof will affect the performance of the program, so do not run with prof or pixie when performing other benchmarking tests.

7.5 Results

The following sections contain the results of benchmarking tests that were performed on ModSAF on October 25, 1993. ModSAF was tested on networked SGIs with 80 meg of memory. The

setup was a safsim running on a local unloaded CPU with very little traffic to the safgui. The vehicles were invincible, had unlimited ammunition, and were continuously moving and shooting.

7.5.1 Network results

This test measured the time spent inside packetvalve i/o calls.

Medium network traffic

SIMNET test

assoc read 266 us/call assoc write 375 us/call

DIS test

udp read 546 us/call udp write 690 us/call

Result: DIS i/o calls take about twice the time as SIMNET i/o calls.

See How to benchmark (see Section 7.3.1 [Network], page 98) for more information.

7.5.2 Vehicle limit results

This test calculated the maximum number of vehicles on a safsim that gives good performance. Refer to How to benchmark (see Section 7.3.2 [Vehicle limit], page 98) to understand what good performance means.

This test was performed with invincible vehicles, unlimited ammo, and continuously moving and firing. Failure criteria was when less than 90% of the ticks in the 67 ring are under 500ms. Setup was safsim running on unloaded CPU connected via network with very little traffic to safgui. Vehicles were added one platoon at a time.

-nonet 28

-simnet -assoc 32

-dis -udp 28

-simnet -udp 32

-dis -assoc 28

Error seemed to be +/- 2 vehicles.

See How to benchmark (see Section 7.3.2 [Vehicle limit], page 98) for more information.

7.5.3 Protocol traffic results

This test measures the percentages of different types of protocol packets that were sent out on the network. The setup was 32 vehicles, configured as in previous tests.

short test (5 mins)

-simnet -assoc -dis -udp

rate: 37 pkts/sec rate: 33 calls/sec

simulation 24% dis 29%

radio 9% po 71%

radioSignal ~0%

po 66%

long test (2+ hours)

-simnet -assoc -dis -udp

rate: 51 pkts/sec rate: ?

simulation 30% dis 43%

radio 11% po 57%

radioSignal ~0%

po 59%

See How to benchmark (see Section 7.3.3 [Protocol family traffic], page 99) for more information.

7.5.4 Thoughts

This is a list of some general thoughts about the performance of ModSAF as of November 1993.

- Doesn't seem to spend much time in network code.
- Number of vehicles able to be simulated seems to be terrain and geometry specific.
- When the sim code is run on a machine with only 64Mb, we don't swap. Performance is the same as a machine with more memory, at least at the number of vehicles we are currently able to simulate.
- Tick length can sometimes get very long and then improve dramatically.
- Running sim under IRIX 5.1.1.1 on an Onyx didn't improve performance.
- More complex behaviors (such as occupying a position) will probably drop the number of vehicles we can simulate.
- Protocol family traffic probably varies over time, and single percentages are of dubious use.
- Using simnet protocol seems to yield better performance. Additionally, the drain device used for assoc seems to do i/o twice as fast as using udp.